UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

Security assessment for automotive controllers using side channel and fault injection attacks

A Master's Thesis
Submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by
Santiago Córdoba Pellicer

In partial fulfilment
of the requirements for the degree of
MASTER IN ELECTRONIC ENGINEERING

Advisor: Francesc Moll Echeto

Barcelona, February 2018

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

telecom BCN

Title of the thesis: Security assessment for automotive controllers using side channel and fault injection attacks

Author: Santiago Córdoba Pellicer

Advisor: Francesc Moll Echeto

Abstract

Embedded security is nowadays a hot topic. With the arrival of Internet of Things and the increasing demand of connectivity for embedded systems in many industrial markets, including automotive systems, security has become an important factor in product design. This thesis is aimed to test the security capabilities of automotive electronic devices, using physical attacks known as fault injection. Although other industries have been using countermeasures against physical attacks for decades, these are rarely used in automotive embedded systems. Automotive industry efforts have been focused in improving safety and reliability (e.g. ISO 26262 ASIL certification) instead of security. Previous research proved the risk of fault injection attacks on automotive SoCs, but these works were limited to small testing applications running on evaluation boards and not real automotive systems. The current work aims to assess the security of off-the-shelf automotive systems running real applications. More specifically, fault injection attacks are used to bypass the authentication mechanism of the Unified Diagnostic System (ISO 14229) present in two different commercial car dashboards. The findings are exposed in order to suggest design improvements and recommendations for a more secure automotive embedded systems and SoCs.

Keywords **- Fault Injection; Automotive; ECU; UDS; ISO 14229**

Para todos aquellos que me han acompañado en este viaje:

a mi familia, desde siempre y hasta siempre;
a mis amigos, a quienes la distanca nunca ha importado;
y a los errantes y nómadas de este mundo, estén donde estén.

S.

Contents

1 Introduction

Vehicles have changed considerably in the last decades. Automotive design is abandoning traditional mechanical solutions for newer electronic based solutions. Modern cars include all type of electronic systems implementing a wide variety of functionalities, from switching lights to control the injection of fuel in the engine. These electronic systems have great advantages and they make possible to design more efficient, more comfortable and safer cars. With a notable increase in the number of electronic controllers in vehicles, electronic design is nowadays an important stage on automotive industry.

Electronic design has many aspects to consider in its work-flow. Every design has a certain purpose in mind, usually in terms of performance, power saving and cost. In industrial and some consumer designs, reliability is a critical aspect to consider. Reliability is specially important when a malfunction on the design may lead to a dangerous and harmful situation. It is not hard to see why an automotive design would have strict reliability requirements: malfunctions in a critical electronic system of a car can put in risk the car passengers and other people in the car surroundings.

Along with the reliability requirements, there is another aspect in electronic design that is less known and not frequently considered: security. Designs with security in scope have protection from undesired modifications and accesses that may as well lead to a undesired or dangerous situation. A poor security automotive electronic design may be exposed to attacks that may lead to severe consequences:

- Firmware extraction: this implies the extraction of the program running in the digital system, which may contain proprietary technologies, an important asset for manufacturers, leading to industrial espionage and unfair competitive edge. The analysis of the extracted firmware can reveal also exploitable vulnerabilities that the attacker may use to compromise the system in a different way. For example, a vulnerability found in a cryptographic function from the system, discovered using firmware extraction and analysis, could lead to an exploit that would allow an attacker to unlock the doors of the car without the genuine keys.

- Firmware modification: after a firmware extraction attackers could make a firmware modifications. Applying patches to the firmware of a automotive electronic controller is one of the objective of a car tuner attacker, motivated by a possible increase of performance in the vehicle by altering the software. These modifications are a great risk to the integrity of the car and could potentially lead to accidents. Furthermore, it is possible to modify the software of a car controller intentionally and inflict damage to the vehicle and its occupants by the means of sabotage.

- Runtime control and access to debug interfaces: these are usually intermediate steps in the way to the previously mentioned situations. Having runtime control and access to debug interfaces could also help in the firmware download, analysis and modification. Having a running target

is very helpful in order to reverse engineer a firmware image, and also to develop patches and modifications.

Along with this scenarios, other security related issues may happen. An example of this is the violation of passenger's privacy, as an attacker could compromise the convenience system in which the hands-free phone call system is built in, listening to the phone calls made or answered from the car, or even listening what the car microphone is recording. Moreover, an attacker could target the GPS navigation system and track a given vehicle, gathering precious information about the passengers' location.

1.1 Motivation and Objectives

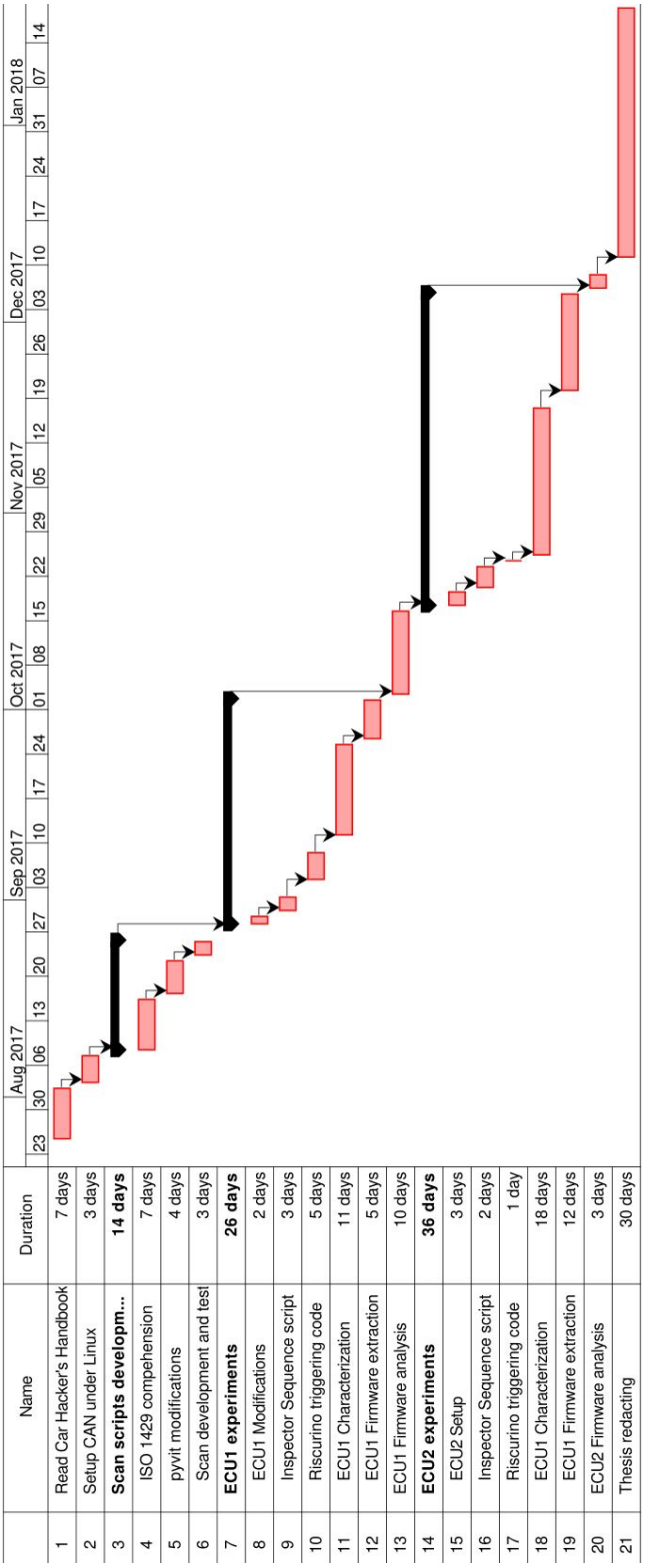
The objective of the present work is to assess the security of automotive systems using fault injection (FI) techniques. Academic research regarding fault injection in automotive systems with security implications in scope is almost nonexistent. Previous work regarding security in automotive hardware make use of FI to unlock access to debug interfaces [1] and then compromise the system. Nevertheless, these experiments are performed under laboratory conditions, commonly using a development board as targets, whereas the present work uses commercially available electronic control units (ECUs) as targets.

The present work aims to discern whether Fault Injection attacks are feasible or not in commercially available out-of-the-shelf electronic control units used in nowadays vehicles. For that purpose, the diagnostic services commonly integrated in the ECUs are used. These services are used to obtain the current status of a vehicle and the issues that the electronic system may have self-diagnosed. Diagnostic services are often used also for upgrade and update the firmwares running in the vehicle electronic controllers. Unified Diagnostic Services (UDS, see Appendix A) is a common specification for diagnostic services, present in most modern vehicles. UDS specifies services for firmware upgrading, security access and other security-critical aspects of vehicle diagnostic. This fact makes UDS an excellent target application for assessing the security of automotive controllers by means of fault injection, as bypassing the security measures implemented for UDS using FI techniques could lead to compromise the controller itself.

Lastly, the task organization for this thesis is shown in Figure 1. The initial task list is included, along with some subtask that were defined and added later. The duration of the tasks has been updated to match the final time investment, as no previsions for their duration were made beforehand.

1.2 About Riscure BV

This thesis was developed in collaboration with Riscure BV. Riscure is an independent security test laboratory specializing in security testing of products based on smart card and embedded technology. Riscure evaluates the security of chip technology and embedded/connected devices that are meant to operate securely in any environment. We are the leading security test lab for chips



	Name	Duration
1	Read Car Hacker's Handbook	7 days
2	Setup CAN under Linux	3 days
3	**Scan scripts developm...**	**14 days**
4	ISO 1429 compehension	7 days
5	pyvit modifications	4 days
6	Scan development and test	3 days
7	**ECU1 experiments**	**26 days**
8	ECU1 Modifications	2 days
9	Inspector Sequence script	3 days
10	Riscurino triggering code	5 days
11	ECU1 Characterization	11 days
12	ECU1 Firmware extraction	5 days
13	ECU1 Firmware analysis	10 days
14	**ECU2 experiments**	**36 days**
15	ECU2 Setup	3 days
16	Inspector Sequence script	2 days
17	Riscurino triggering code	1 day
18	ECU1 Characterization	18 days
19	ECU1 Firmware extraction	12 days
20	ECU2 Firmware analysis	3 days
21	Thesis redacting	30 days

Figure 1: Gantt chart.

and set-top boxes deployed in the pay-TV industry. Riscure is also international market leader in providing test equipment for side channel robustness of chip technology. Riscure laboratory equipment is used by chip manufacturers, government agencies and security test laboratories around the world. In order to achieve the main objectives, it is needed experience and background in automotive embedded systems, along with deep knowledge in fault injection techniques and adequate equipment to perform experiments, and Riscure provided the needed experience, supervision and support during the research.

2 State of the Art

This section will discuss the techniques known as Side Channel Attacks, used in this work to assess the security of complex electronic automotive systems.

2.1 Side Channels

When evaluating a secure application, even when the application itself is considered secure, the implementation of it is not implied to be secure. The way an application is implemented on a device can make it vulnerable. Side channel attacks rely on physical aspects of the implementation, instead of theoretical vulnerabilities of the implemented application itself. The name of side channel is given because of the use of unintended physical channels, intrinsic to the implementation, to interact with the application itself. Different ways of exploiting the implementation aspects of a target have been proposed in [2].

In [3] there are described two common criteria to categorize these attacks. The first criterion is whether the attack is passive or active:

- Passive attacks: In this kind of attack, the device under attack runs mostly or entirely under normal circumstances. The device is compromised by observing physical properties of the device operation (e.g. execution time, power consumption, electromagnetic emissions).
- Active attacks: In these attacks the device, its inputs, and/or its environment are manipulated in a way such the device behave abnormally. The device is then compromised exploiting undefined or undesired behavior.

The second criterion for categorizing SCA is the interface used to attack the device. A device may have several interfaces, logical or physical. Based on the interface used and the easiness to access it, it is possible to distinguish between invasive, semi-invasive and non-invasive attacks:

- Invasive attacks: In these kind of attacks there is essentially no limit in terms of interfacing and exploiting. Invasive attacks typically start with the decapsulation of the device, and followed with high technology passive or active techniques.
  Among the passive invasive attacks, microprobing is one of the most common. Invasive active techniques worth mentioning are modifications using laser cutters or focused ion beams.
  The equipment needed for this attack is expensive, and they are rarely performed.
- Semi-Invasive attacks: In these attacks, the target device is usually also decapsulated, but in contrast with invasive attacks, in semi-invasive attacks no direct electrical contact is made.
  Passive semi-invasive attacks are usually related with reading memory cells without probing. Active semi-invasive attacks are intended to introduce faults in the device using electromagnetic fields or light.
  These attacks typically do not require equipment as expensive as invasive attacks, but the effort and time needed to conduct them are relatively high.

- Non-Invasive attacks: In these kind of attacks the device is attacked as is, without modification and exploiting only the directly accessible interfaces. Passive non-invasive attacks are usually known as side-channel analysis, and the most popular types are timing analysis attacks, power analysis attacks and electromagnetic (EM) analysis attacks. The idea behind these techniques is to compromise a device by measuring its execution time, its power consumption or its electromagnetic emissions, relating the measured traces to the operation running in the target. On the other hand, active non-invasive attacks are frequently known as fault injection (FI) attacks. The main objective for FI attacks is to introduce a fault into a device that results in unexpected or undesired behavior without the need of heavily modify the target. The most used techniques to inject faults are clock glitching, power glitches, electromagnetic fault injection (EMFI) and Optical fault injection.
  Most of these attacks can be performed with inexpensive equipment in comparison with invasive attacks and even with semi-invasive attacks. Hence, non-invasive attacks are a major threat to devices security.

2.2 Fault Injection Techniques

As discussed, FI attacks are active side channel attacks. The main idea of fault injection attacks is to deliberately put the device under stress by perturbing its working conditions through different methods [4].

2.2.1 Clock Glitching

This perturbation model is based on a temporal and controlled overclocking of the device. For devices supplied with an external clock, a maximum clock frequency f_{max} is typically defined. The minimum period T_{min} equals the reciprocal of f_{max}. As long as the injected clock glitch period $T_g >= T_{min}$, no erroneous behavior can be observed, but if $T_g < T_{min}$, the probability for observing an erroneous behavior increases. The reason for the faults is a timing violation. Each combinational path in a circuit has a specific propagation delay (T_P), which describes the time interval between changing the input and providing a valid output value. Registers at the output of the combinational logic block sample the value, typically at the positive clock edge. A representation of

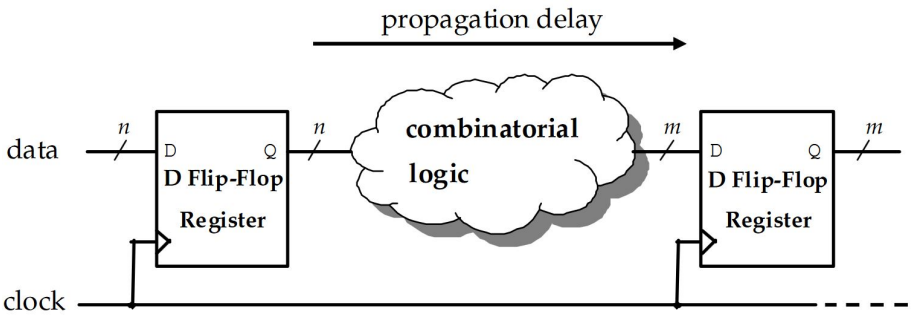


Figure 2: Simple representation of a typical digital path [5].

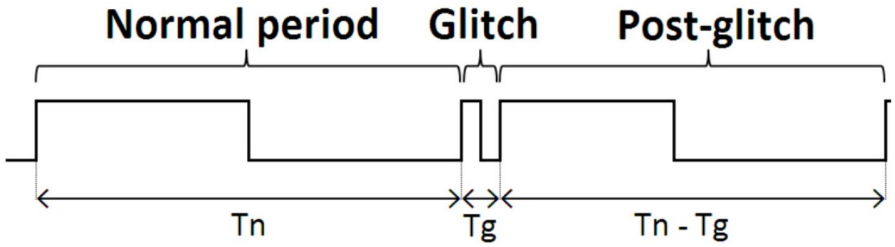


Figure 3: Clock glitch example.

this is provided in Figure 2. If the positive clock edge arrives before the output of the combinational logic has settled to a stable value due to a timing violation enforced by a clock glitch ($T_g < T_P$), the registers store erroneous values leading to faulty computations as a consequence [6].

For achieving this, a clock period such as $T_g << T_{min} < T_n$ is inserted, with the intention that it causes a transient malfunction of the target as the result of a critical path timing violation. The clock signal is usually generated as described in Figure 3. Notice that after injecting a glitch, the following clock period is reduced from T_n to $T_n - T_g$. However, given that $T_g << T_n$, this post-glitch period does not affect the normal behavior of the target [7].
Obviously, these kind of attacks require access to the target device clock signal, and modern microcontrollers usually synthesize different internal clock signals for several domains from an external reference clock, so clock glitching is not possible, at least in a non-invasive attack.

2.2.2 Voltage Glitching

This perturbation model is based on a temporal and controlled underpowering of the device. Underpowering means to reduce the value of the supply voltage of the attacked device below the minimum value the device is specified for.

Consider a simple CMOS inverter gate as shown in Figure 4. According to [8], it is possible to recall the propagation delay t_{pLH} (Equation 1) as a function of the power supply by means of a first order analysis of the inverter dynamics:

$$t_{pLH} = \frac{C_L[\frac{2|V_{th,p}|}{V_{DD}-|V_{th,p}|} + ln(3 - 4\frac{|V_{th,p}|}{V_{DD}})]}{\mu_p C_{ox}\frac{W_p}{L_p}(V_{DD} - |V_{th,p}|)} \quad (1)$$

where V_{DD} is the power supply voltage, C_L the load capacitance, $V_{th,p}$ the PMOS threshold voltage, μ_p the holes mobility, C_{ox} the gate oxyde capacitance and (W_p/L_p) the aspect ratio of the PMOS. A similar equation for t_{pHL} may be derived from Equation 1 by substituting the parameters related to the inverter NMOS transistor for those related to the PMOS. Note that t_{pHL} and t_{pLH} may have different values. Hence, the data propagation time through the inverter (and through any logic block) depends on the handled data: the propagation

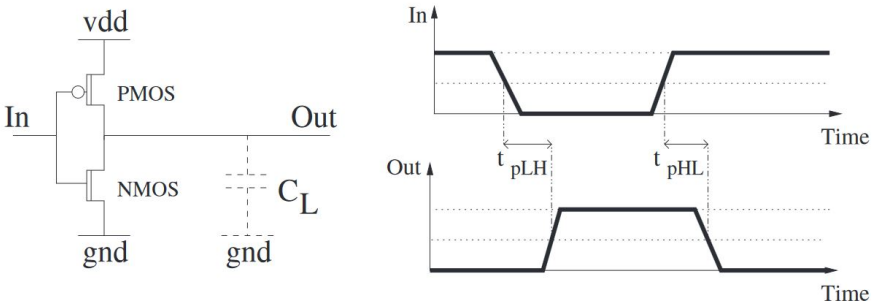


Figure 4: Inverter: architecture and waveforms.

time is data-dependent.

As stated by Equation 1 any decrease of V_{DD} will induce an increase of the propagation delay of the inverter. By extension, the data propagation time through any logic block is increased as long as the IC is underpowered. Hence, underpowering is a valid technique to achieve fault injection by violation of the timing constraints [9]. Due to this fact, similar effects can be achieved when glitching the clock signal and the supply voltage [6].

Voltage glitching attacks usually need minor modifications on the target device [10], such as:

- Power cut: the target IC needs to be disconnected from the on-board power supply, and then obtain control over the target voltage supply.
- Capacitance: the target capacitance impacts the effectiveness of the injected glitch. Therefore, the capacitance is reduced to a minimum by removing as many bypass capacitors as possible from the PCB.

Nevertheless, there are other characteristics present in modern devices that prevent voltage glitching attacks, such as on-die voltage regulators, multiple power domains and active countermeasures such as voltage sensors. The presence of regulators in the IC makes the attack drastically more difficult, as the injected glitch becomes compensated by those regulators.

2.2.3 Electromagnetic Fault Injection

This perturbation model is based on a temporal and controlled electromagnetic pulse over the device. By means of electromagnetic induction, the pulse induces a current in the target. This current momentarily changes the voltages and logic values of the internal gates, and this may lead to a computational fault in the target [11].

Electromagnetic induction is caused by the interaction between electric fields and magnetic fields, according to Ampere's circuital law and the Maxwell-Faraday equations. Alternating the polarity of a magnetic field in time causes an electric field to be generated. The consequence of this is, if a varying magnetic field flux passes through a closed circuit, then a current is generated in

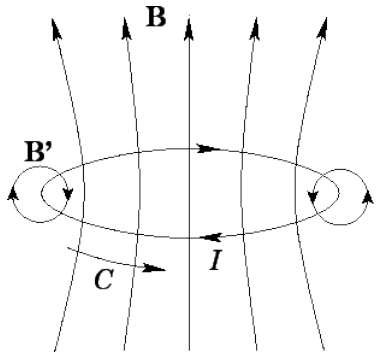


Figure 5: Schematic view of a loop of electrical wire with a current I in clockwise direction The current generates a magnetic field B and B'.

that circuit. Figure 5 shows a schematic view of the effect of a magnetic field on a ring-shaped conductor. If the strength of the flux (i.e. number of B lines crossing the loop area) is changed, then the current I in the ring will change as well. If the polarity of the flux is changed then the direction of the current is switched as well. Similarly, a varying magnetic field can be generated by varying the current in a ring-shaped conductor [12].

The intensity of the magnetic field B generated at the center of a ring-shaped conductor with radius R and constant current I can be calculated using Equation 2:

$$B = \frac{\mu_o \mu_r I}{4\pi R^2} \oint dL = \frac{\mu_o \mu_r I}{4\pi R^2} 2\pi R = \frac{\mu_o \mu_r I}{2R} \tag{2}$$

where μ_o is the magnetic permeability of vacuum and μ_r is the relative permeability of the material the loop is filled with. Homogeneously, the current induced in a loop by a magnetic field B can be calculated using Maxwell-Faraday Equation, described in Equation 3:

$$\oint_{\delta\Sigma} E\, dl = -\int_{\Sigma} \frac{\partial B}{\partial t}\, dS \tag{3}$$

where Σ is the surface bounded by the loop, with contour $\delta\Sigma$ and E is the electrical field. If we assume that the surface does not change (i.e. the conductor loop geometry remains constant in time) we can rewrite Equation 3:

$$\oint_{\delta\Sigma} E\, dl = -\frac{\partial}{\partial t} \int_{\Sigma} B\, dS = -\frac{\partial}{\partial t} \Phi_B \tag{4}$$

where Φ_B is the value for the magnetic flux through Σ.

Equation 4 implies that an electric field can be injected in a closed circuit by varying over time the magnetic flux traversing the surface identified by the circuit itself, hence the time partial derivative term on the right side of the Equation 4. The electric field generates a difference of potentials across different parts of the circuit, which induces currents as the free charges move to follow

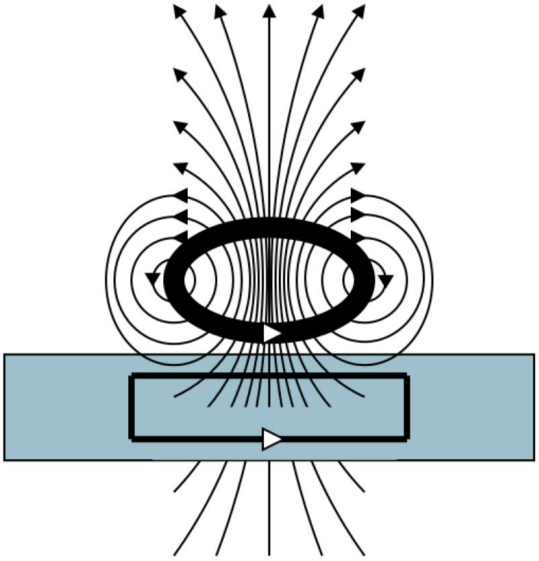


Figure 6: Schematic view of a EMFI device. Black ring represents the coil and gray plane represent the target surface. Black headed arrows represent magnetic field, while white headed arrows represent currents.

the potential gradient. Equation 4 proves that the strength of the induced potential gradient is dependent on the time-derivative of the magnetic flux, and not only the absolute value of the flux itself.

Electromagnetic Fault Injection instruments are designed to provide very sharp variations on the magnetic field $B(t)$, so stronger currents and potential gradients can be induced in the circuit. In order to use these effects to inject faults, EMFI equipment usually is made of a varying current source that injects an abrupt current pulse into a coil that is placed near the target device. Figure 6 shows a schematic representation of such equipment. This sudden current pulse generates a varying magnetic field that is able to induce currents in any closed loop of the device under the coil. The potential gradients may change a transistor from off to on or vice-versa, depending on the type of transistor and the polarity of the voltage glitch introduced (i.e. the inducted current direction, which depends on the coil current direction) [13].

The voltage glitch will only switch the state of one P-channel and N-channel transistor pair and therefore does not cause short circuit between VDD and GND. Eventually, this change in the state can introduce computational faults, resulting in a successfully injected fault. Figure 7 gives an example of a circuit layout. The coil located above M2 induces voltage glitches separately in two loops: the loop outlined in red (center) and the loop outlined in orange (far right). Since the loop area in red is much bigger than that in orange, the loop in red will get most effective glitch [14].

Nevertheless, the usual dimensions for EM probe coils are in the scale of

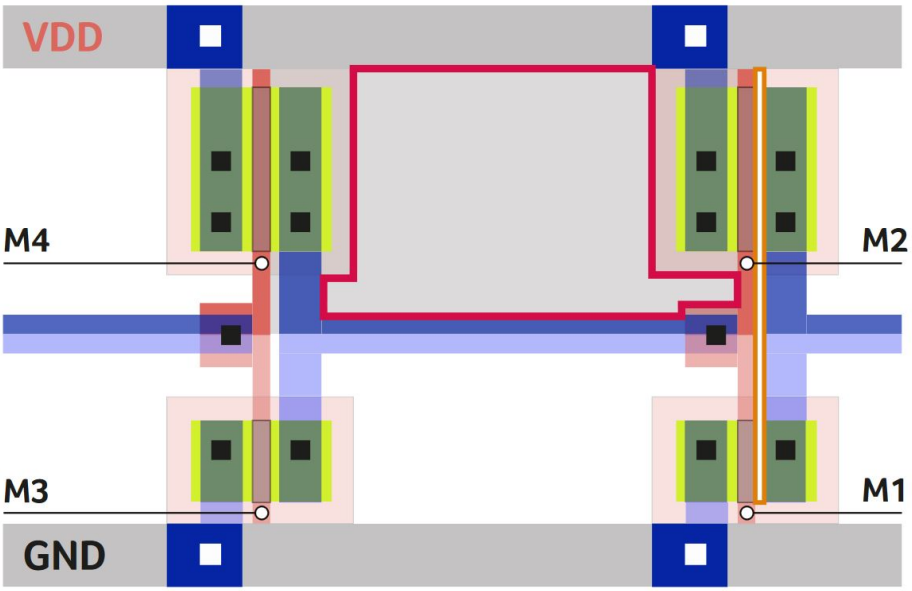


Figure 7: Circuit layout showing the available closed loops in the circuit (red and orange) [14].

millimeters and are not comparable to the scale of the circuit, as the effective coil area is much bigger than a single transistor pair. This means that the effects of EMFI are substantially macroscopic, usually affecting to bigger metallic structures as long parallel buses.

Obviously, this makes EMFI attacks sensitive to location changes. The coil is often placed into a XYZ stage in order to fine tune the coil position over the target device. This sensitivity also gives these kind of attacks the essential advantage to an attacker of targeting a specific part of the IC, and leave the rest unperturbed. This is desirable if detectors and countermeasures are present. The main disadvantage is, then, knowing where the locations of interest are in the whole of the IC surface. It is common to run scans all over the IC surface to characterize its behavior when exposed to EM pulses and then extract conclusions, but those scans can take a long time.

2.2.4 Optical Fault Injection

This perturbation technique is based on the exposure of the IC semiconductor to light, usually by means of a very focused laser beam. The laser photons generate free electrons in the P-channel and N-channel of transistors and, as a result, the conductivity of any transistor inside the laser spot increases and transistors switch to on state. The laser located above M2 and M1 in Figure 8 changes the status of M1 or M2 or both, which depends on the laser wavelength and pulse strength. A description of the relation between laser wavelength, energy and its effect on transistors is described in Figure 9. If conductance of

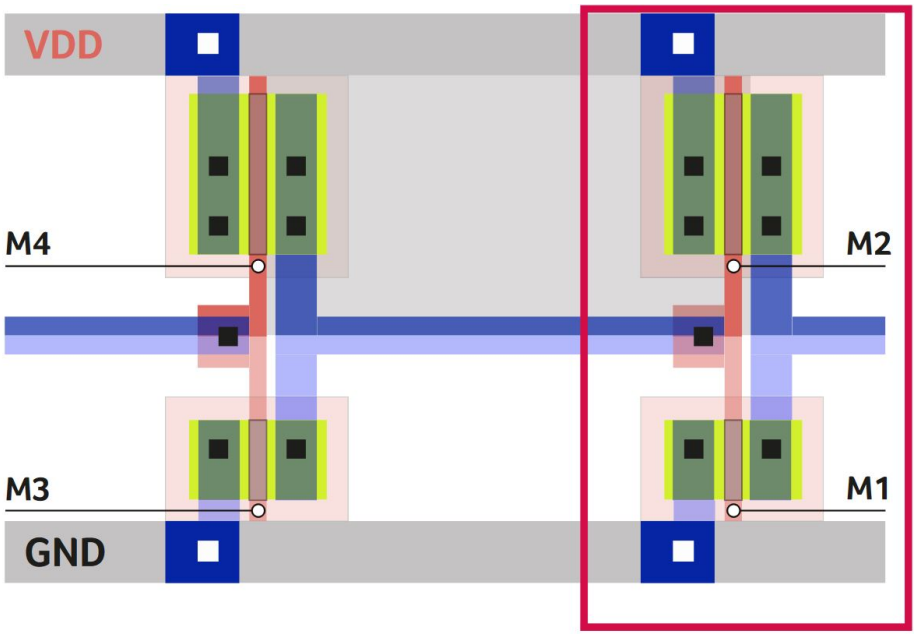


Figure 8: Circuit layout showing an example of the affected transistors by a laser beam in a optical FI [14].

both transistors of a P-channel and N-channel pair happens, it causes a short circuit (i.e. latch-up effect because of the parasitic structures intrinsic to CMOS technology) between VDD and GND that can damage the chip [14].

Optical FI almost always requires the target device to be heavily modified using mechanical or chemical processes until the silicon parts of it become visible. As mentioned earlier in this chapter, this means that Optical FI is a semi-invasive active side-channel attack, instead of a non-invasive. Even considering it, Optical FI is often used in security labs to assess the security capabilities of a secure application IC.

2.3 Effects of Fault Injection

Once explained the techniques to inject faults in a target, the effect of those faults should be discussed. When attempting to inject a fault, there are several possible results:

- Unsuccessful glitch: the attack has no effect on the target and it behaves normally. This means that the attack failed.
- Crash: The glitch affects the target but causing unstable behavior. In some architectures this can mean a processor exception, where the target is held in an infinite loop as a result of an exception handler.
- Reset: The glitch affects the target but causing a reset in the device.

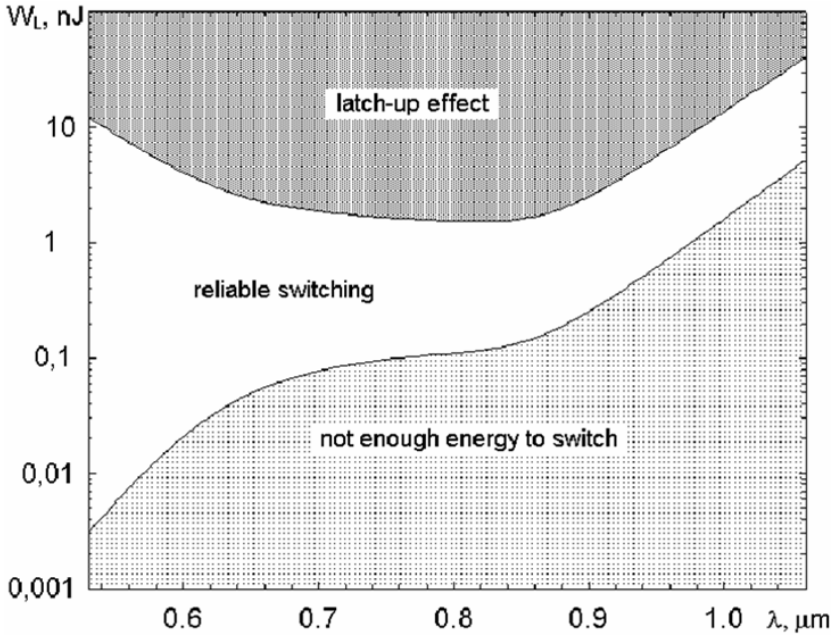


Figure 9: Relationship between laser wavelength, pulse energy and its effect in transistors [15].

- Successful glitch: The glitch affects the target resulting in the expected behavior the fault was meant to produce.

Also, we can classify the effects of the fault according to its duration in time:

- Transient: the effect only lasts while the perturbation is applied. E.g. a fault that produces a wrong Arithmetic Logic Unit (ALU) result, but the following results for the same ALU are correct, even without reseting the device.
- Temporary: the effect lasts until next device reboot. E.g. a change in a peripheral control register bit, that will only restore its original value after a reset.
- Permanent: the effect lasts permanently. E.g. a corrupted writing operation into non-volatile memory, or any other physical change in the actual device that may affect the target functionality and is not restored after a reset.

2.3.1 Fault Models

There are many ways a fault can affect a target, as studied in previous works such as [16], [17] or [18]. The different ways a fault may affect a device are modeled into what is known in the literature as fault models.

Memory related: Memory can be viewed as consisting of two parts: memory itself and the memory bus. The task of the memory bus is to transport data

and instructions to and from memory. The bus is usually composed of the data and the address bus:

- Address bus: selects the memory location to read from or the memory location to store to.
- Memory data bus: transports the data stored at these locations.

Both can fail independently of each other. If there is a fault on the data bus, incorrect data are presented. If the address bus is glitched, the data presented will be correct but from/to the wrong location. Identifying which faults occurred will be hard, especially if both occur at the same time.

Registers related: Registers are an important part of the execution. Target devices usually contain both special use registers and general purpose registers. They can be glitched mainly causing a static corruption on the contents. Then, depending on the affected register it may lead to wrong results (e.g. a general purpose register was corrupted and then used in an ALU instruction) or to more sophisticated faults (e.g. corrupting the program counter register leads to a corrupted jump, corrupting the stack pointer register leads to incorrect data retrieving from the stack).

Instructions related: Instructions are commonly categorized into three groups: flow control, memory operations and ALU operations. Flow control operations include instructions like branch and jump. Memory operations either store or load data from memory. Last, ALU operations are computational operations. Instructions and the way they are executed are highly architecture dependent. For example, in a Von Neumann architecture, instructions have a four phases cycle for execution:

1. Fetch phase: in this phase the instruction is retrieved from memory. Here memory related faults apply, as discussed in 2.3.1, so corrupted instructions can be fetched, leading to different instructions or even illegal instructions being fetched.
2. Decode phase: in this phase the fetched instruction gets decoded, assigning meaning to the data retrieved. A fault in this phase changes the way the instruction is decoded, leading to different instructions or even illegal instructions being executed. Thus, it is very hard to distinguish this faults from the fetch phase faults.
3. Execution phase: Once decoded, the instruction gets executed. Faults introduced during the execution phase would manifest themselves by computing an incorrect result.
4. Store phase: The store result phase of the instruction execution cycle stores the result from the executed instruction in a register or memory. A fault during this phase would manifest itself as a corrupted or un-updated value being stored.

Corrupted instructions may transform themselves into illegal instructions, causing a crash or reset; or may transform into legal (but different) instructions.

Instruction corruption

`MOV R0, R1`	`11100001101000000000000000000000`
`MOV R0, R2`	`11100001101000000000000000000010`

`MOV R0, R1`	`11100001101000000000000000000001`
`STR R7, [R7, #16]`	`11100101100010010111000000010000`

Instruction skipping

`MOV R0, R1`	`11100001101000000000000000000001`
`MOV R1, R1`	`11100001101000000010000000000001`

`MOV R0, R1`	`11100001101000000000000000000001`
`MOV R6, R6`	`11100001101000000110000000000110`

Figure 10: Examples of instruction faults in ARM instruction set. Left column is mnemonic assembly code, right column is the binary machine code related to that instruction. Bits affected from a fault are showed in red [24].

The result of this depends on the effects of the fault into the instructions, and goes from a mere change in the operands to a totally different instruction. Also, instruction skipping may happen as a result of instruction corruption, as a given instruction can be transformed by means of a fault into a useless instruction. Figure 10 shows an example of these effects in ARM architecture instruction set.

These faults can have important effects in the code running in a device, such as skipping a security check or providing a wrong result in a cryptographic operation. Fault attacks pose a serious threat in cryptographic systems and they can be applied on a multitude of physical implementations of various cryptographic algorithms like AES [19], DES [20], ECC [21], or RSA [22][23]. Differential Fault Analysis are a set of techniques to extract the secrets from these cryptographic algorithms injecting faults and recovering faulty results. Comparing these results with the correct result (i.e. non injected fault result) may lead to expose the internal state of the algorithm primitives and therefore to expose the secret key. It is also very common to override the debug interfaces settings in a given target device by means of fault injection. In [1] it is described the process to unlock the JTAG interface of several SoCs using fault injection.

3 Methodology

In this section it will be described the experiments performed during the research of this work.

3.1 Objectives

As introduced in Chapter 1, the objective of the experiments is to assess the security of automotive systems using fault injection techniques. Previous works regarding security in automotive hardware make use of FI to unlock access to debug interfaces and then compromise the system [1]. The present work takes a different approach: using the automotive diagnostic services built in modern Electronic Controller Units (ECUs) as a attack vector instead. Furthermore, previous works experiments are performed under laboratory conditions, commonly using a development board as targets, whereas the present work uses commercially available ECUs as targets.

3.2 Targets

The experiments will have two different target ECUs. Both are known in automotive industries as "instrument cluster" or "dashboard". This units are responsible for giving the driver all kinds of information about the state of the car. Some examples are the current speed, the engine RPMs or the current fuel consumption and level. They also display information about the possible problems the car may have, known as Malfunction Indicator Lights (MIL). Figure 11 shows a generic dashboard ECU example.

For confidentiality reasons, the assembler of the ECUs and the manufacturer of the target ICs are not disclosed. Both targets, from now referred to as $ECU1$ and $ECU2$ are from the same major automotive manufacturer, being $ECU1$ from B-segment 2010 model, while $ECU2$ is from a C-segment 2015 model[1]. Both have very similar hardware, to the point of having microcontroller units (MCUs) from the same manufacturer and within the same product family, but from different generations: $ECU1$ has an older version of the MCU whereas $ECU2$ is from a newer generation MCU.

As a side note, datasheets were found online for $ECU1$ but not for $ECU2$. The information provided by the datasheet of the MCU for the $ECU1$ made possible to modify and attempt voltage fault injection in this target. Unfortunately the newer MCU in $ECU2$ is not pin compatible and therefore voltage glitching becomes a more time-consuming task for this other target: it is needed to reverse engineer the PCB, tracing all on-board power supply lines, then check whether they are connected to the MCU or not, and finally do the modifications needed. For this reason electromagnetic fault injection was used with $ECU2$.

[1] Vehicle segments are a car classification defined by the European Commision, each segment constituting distinct product markets. More details in `https://en.wikipedia.org/wiki/Euro_Car_Segment`

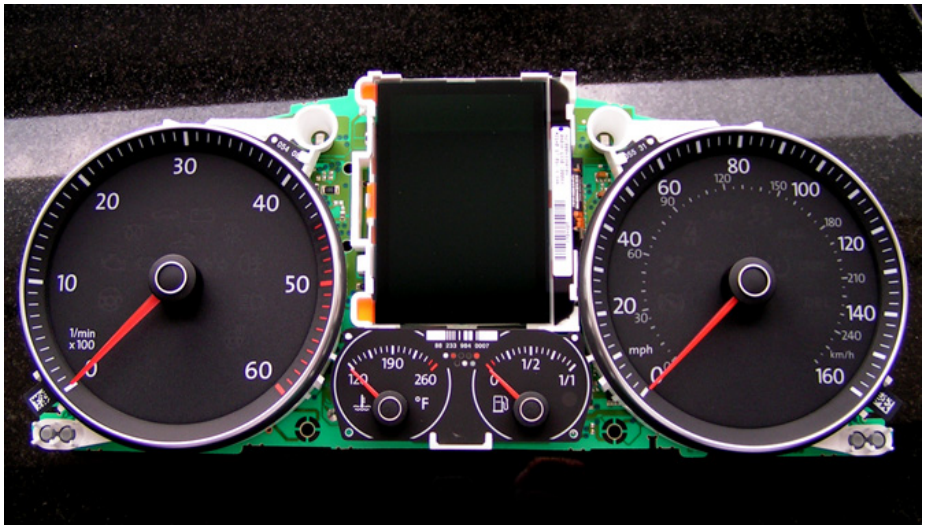


Figure 11: Unassembled instrument cluster board example, with dials and a large LCD screen.

The general architecture of both ECUs is basically the same: an MCU with CAN peripherals, stepper motor control circuitry for the dial instruments, and a liquid crystal display connected using a dedicated LCD controller.

3.3 Fault injection setup

This subsection is aimed to describe a generic setup for fault injection campaigns. A fault injection setup needs several components:

- Target: target itself may need to be modified depending of the needing of the experiment. Alongside with power cut and capacitance reduction modifications (see Section 2.2.2), it is needed to get access to a communication interface (e.g. CAN bus, UART) and to a reset line or mechanism.

- Communication: it is needed a way of communication between the target device and the setup controller. This communication interface is used for preparing the target before the perturbation and receiving feedback from the glitch effect. E.g. using an UART port to send a command to the target and receive a response to that command: if the response is not the usual, a fault was injected; if it is the usual, then the glitch did not affect the target in any expected way. Note that a fault could still be injected, but in a way that is not detectable or evident only by evaluating the output response.

- Reset: glitching attempts very often end putting the target device into a unrecoverable state (i.e. crash results, see Section 2.3), and a reset is needed to set the device back to an usable state. It is also recommendable, if possible, to perform a reset every time before a glitch attempt, to discard any non-observable or non-evident fault that could have been injected in the previous attempt. Unfortunately, some devices have a long boot time,

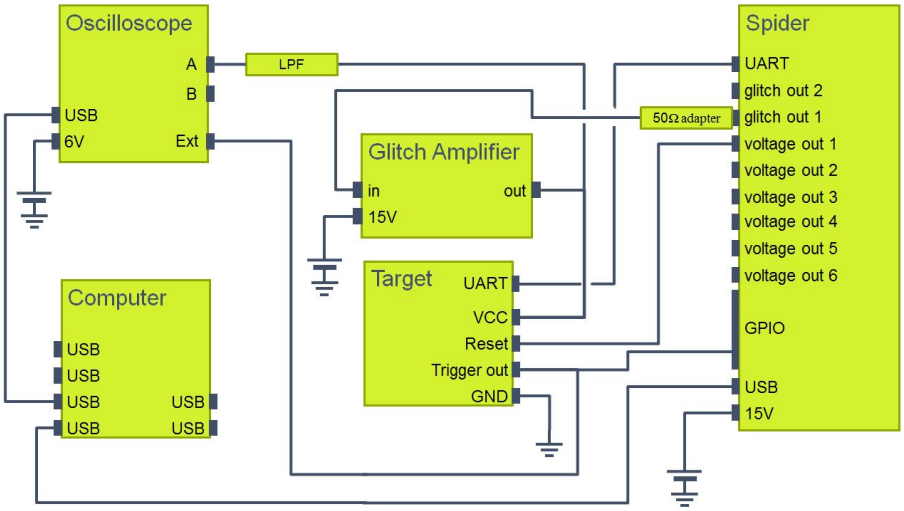


Figure 12: Schematic view of a typical voltage fault injection setup.

and performing a reset before every glitch attempt is not feasible due to the long time it adds to the campaign.

- Trigger: this is, perhaps, one of the most important parts in a FI setup. The trigger allows to synchronize both the target and the glitching equipment. A good and stable trigger makes much easier a FI campaign. For characterization and other experiments it is common to use a GPIO port to set a flag before the target starts running the operation under attack. In other experiments this is not possible, so the trigger is then derived from other parts of the setup, such as communication interface activity or more advanced triggering based on side channel analysis and pattern detection [25].

- FI and control equipment: different perturbation equipment is needed depending on the fault injection technique used. For a voltage glitching setup a controlled voltage source is needed, usually along with a buffer amplifier (i.e. voltage follower). For EMFI setups a EM pulse generator is needed.
  Besides the perturbation equipment, a controller is needed. This controller is the responsible for setting the perturbation parameters, arming the target device (i.e. sending a command or request) and evaluating the response.

Extra equipment can be used, such as an oscilloscope or other measurement devices, to troubleshoot the rest of the setup. Figure 12 shows an example of a typical fault injection setup. In this setup, Spider [26] is used as the perturbation device, communication device and reset mechanism. Triggering signal is generated by the target itself. A glitch amplifier (i.e. voltage follower buffer) is also included. Finally, all the setup is controlled by a computer and monitored using an oscilloscope.

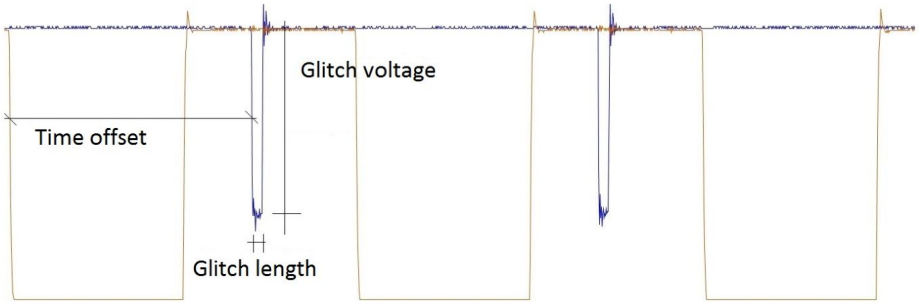


Figure 13: Example of voltage glitching measurements of clock signal (orange) and voltage supply signal with two glitches (blue).

3.4 Perturbation parameters

Perturbations are described using what is known as perturbation parameters. With these parameters, a perturbation can be modeled in terms of timing and energy [10]:

- Glitch delay: The amount of time between the trigger is set and the glitch is injected. This parameter makes triggering crucial, as an incorrect timing in the perturbation can have drastic effects in the glitch result when targeting an specific operation, and also to the reproducibility of the perturbation.
- Glitch length: the amount of time the perturbation is applied to the target, i.e. the amount of time the target is exposed to underpowering, EM pulse or laser burst during the perturbation.
- Glitch amplitude: this models the intensity of the perturbation, i.e. how many volts are dropped to underpower the target, or the intensity of the EM pulse or laser burst the target is exposed to.

Figure 13 shows an example of voltage glitching signals with remarks on its perturbation parameters. Note that when selecting voltage glitching perturbation parameters it is also possible to select the normal (i.e. unperturbed) voltage. It is desirable to set this unperturbed voltage as low as possible (but within operative values) to reduce the impact the capacity has in the glitch timing. This is because glitch falling time depends on capacitance and voltage difference. Like this, there are also other parameters specific to other FI techniques, such as XYZ coordinate for EMFI or Optical FI.

These parameters also make a certain perturbation reproducible, given that the rest of conditions are kept, e.g. the trigger is asserted at the same instant every attempt, or the XYZ reference point has not changed.

3.5 Experimental setup

This subsection contains the description of the experimental setups for both targets $ECU1$ and $ECU2$. Both setups are similar, but each target has been

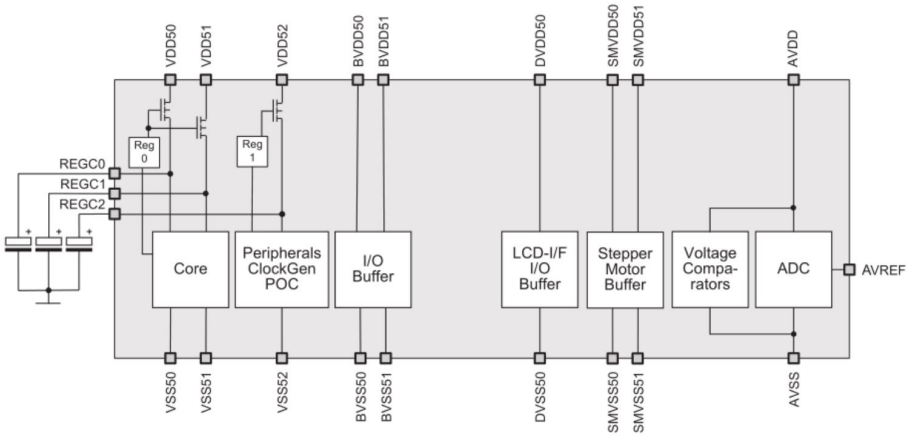


Figure 14: *ECU*1 MCU datasheet figure related to power supply pins.

evaluated using a different fault injection technique, as mentioned in Section 3.2.

3.5.1 *ECU*1

This dashboard was disassembled to expose the PCB and search information about the main MCU in the board. The pinout for the ECU was easily found online, with the location of the power supply and CAN bus pins in the main header.

After identifying the MCU and its datasheet, its power supply pins are located. Thus, voltage glitching is considered. This specific IC has on-die voltage regulators, which adds complexity to the attack, but a workaround is made. Figure 14 shows a capture from the *ECU*1 MCU datasheet. It shows the power related pins and their function within the IC, and the presence of two internal voltage regulators, `Reg0` and `Reg1`. If a voltage glitch is to be injected through pins `VDD50` or `VDD51` with the intention of causing a fault in the core block, `Reg0` would compensate the glitch. Instead, voltage glitches are injected using `REGC0` and `REGC1` pins. These pins, originally intended to install decoupling capacitors for the `Reg0` output voltage, are directly connected to the core block, allowing to bypass the on-die regulator. Furthermore, pins `VDD50` and `VDD51` were lifted (i.e. disconnected from the PCB), decoupling capacitors connected to pins `REGC0` and `REGC1` were removed, and the needed power supply for the core block (nominally 2.5V) was supplied using those pins. Also, using the data provided by the datasheet, MCU reset pin is located and made accessible.

Note that there is no interest in modifying the pins associated with `Reg1`, since this regulator is for peripherals and clock related circuitry and the target is the core itself.

For communicating with the ECU, an extra board is used. It is a Arduino-like custom board, based on a Atmel XMEGA 128A4 microcontroller with two

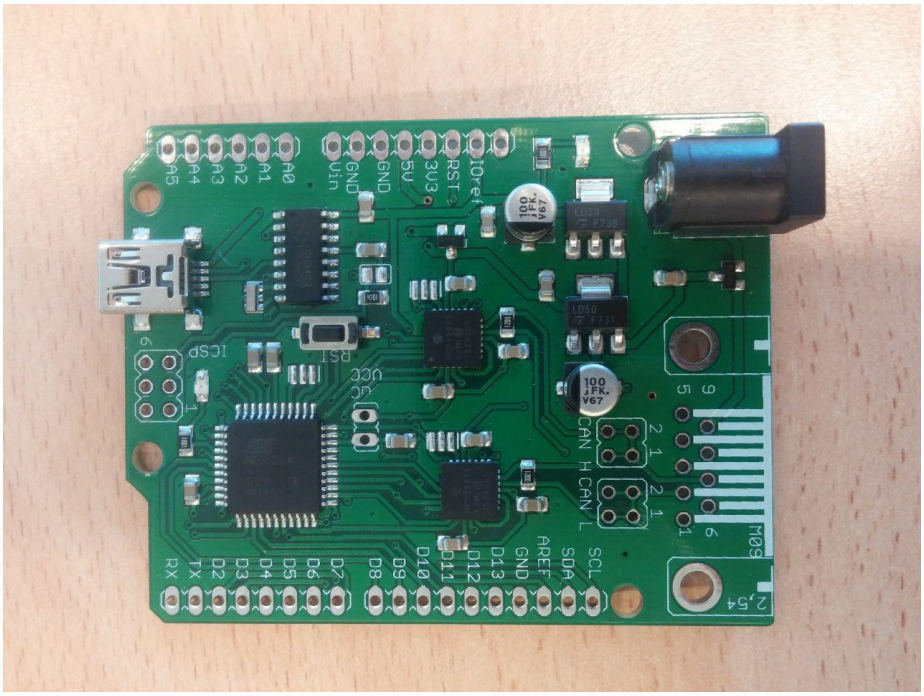


Figure 15: Riscurino: Automotive-oriented MCU development board with Arduino format.

Microchip MCP25625 CAN controllers connected using I^2C. Figure 15 shows a picture of the board. This board runs a program that translates commands from its serial port to its CAN bus, and also asserts a pin when the last command was sent. Since it is not possible to run custom code in $ECU1$ its impossible to modify it to trigger our setup, so this other approach is taken. Algorithm 1 shows a pseudo-code representation of the main function running in the auxiliary board, embedding the communication and triggering.

As for FI equipment and control, Riscure commercial tools are used: Inspector [27] and VC-Glitcher [28]:

Inspector is the software backend for controlling the setup. It includes highly customizable software-controlled trigger and perturbation parameters such as glitch and pulse length, pulse repetition, and voltage level. The software presents the results in a detailed log [27]. Figure 16 shows an example of an Inspector run log, showing expected behavior (green), target resets (yellow), and unexpected behavior (red), along with the glitch parameters that caused the result.

VC-Glitcher is the hardware used to provide an arbitrary voltage signal (including glitches) to the target. It consists of a Field-Programmable Gate Array (FPGA) and Digital-to-Analog Converter (DAC). The DAC is able to provide an arbitrary voltage signal between -4 V and 4 V with a 4 nanosecond pulse resolution. The voltage signal is used to provide power to the the MCU core

Data: command_list with N commands, received from UART
Result: response to Nth command
init_platform;
foreach *command in command_list* **do** CAN send *command*;
if *command is last command* **then**
    assert trigger;
    save response;
else
    discard response;
end
UART send response;
;

Algorithm 1: Basic algorithm for communication and triggering from experimental setup

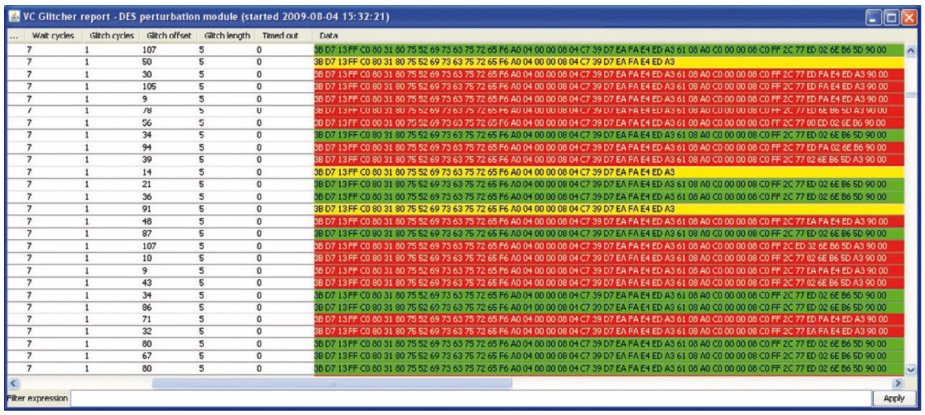


Figure 16: Inspector log screen example for a FI campaign [28].

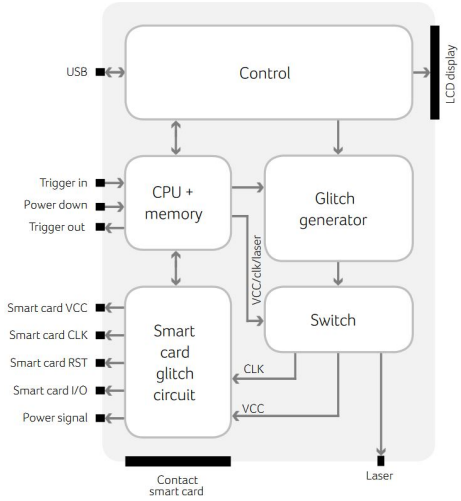


Figure 17: Conceptual overview of VC-Glitcher [28].

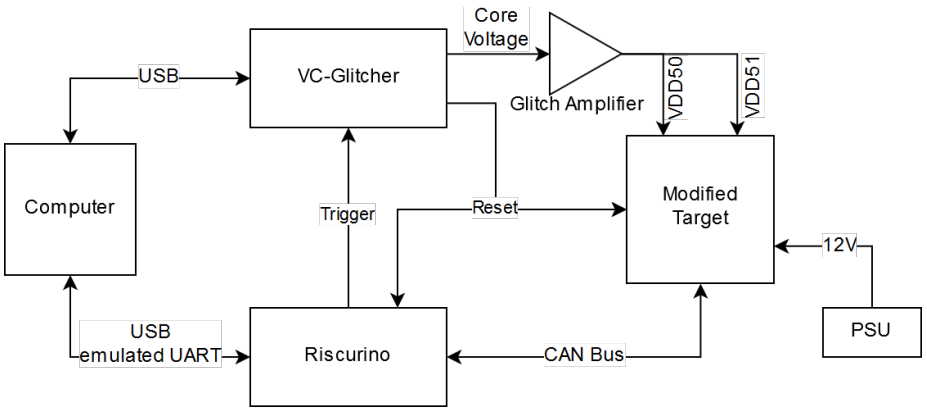


Figure 18: Schematic for $ECU1$ setup.

using a glitch amplifier (i.e. high bandwidth voltage follower amplifier) that can output a current up to 1.5 A. It is also the responsible of asserting the reset line for the whole setup [10][28]. Figure 17 shows a schematic view of the tool. Note that the interfaces used for this setup are labeled as smart card ports, but it is trivial to connect them to a embedded target.

Figure 18 shows an schematic view of the complete setup for $ECU1$. Figure 19 shows an actual photography of the setup.

3.5.2 $ECU2$

This dashboard was also disassembled to expose the PCB and search information about the main MCU in the board. The pinout for the ECU was easily found online, with the location of the power supply and CAN bus pins in the

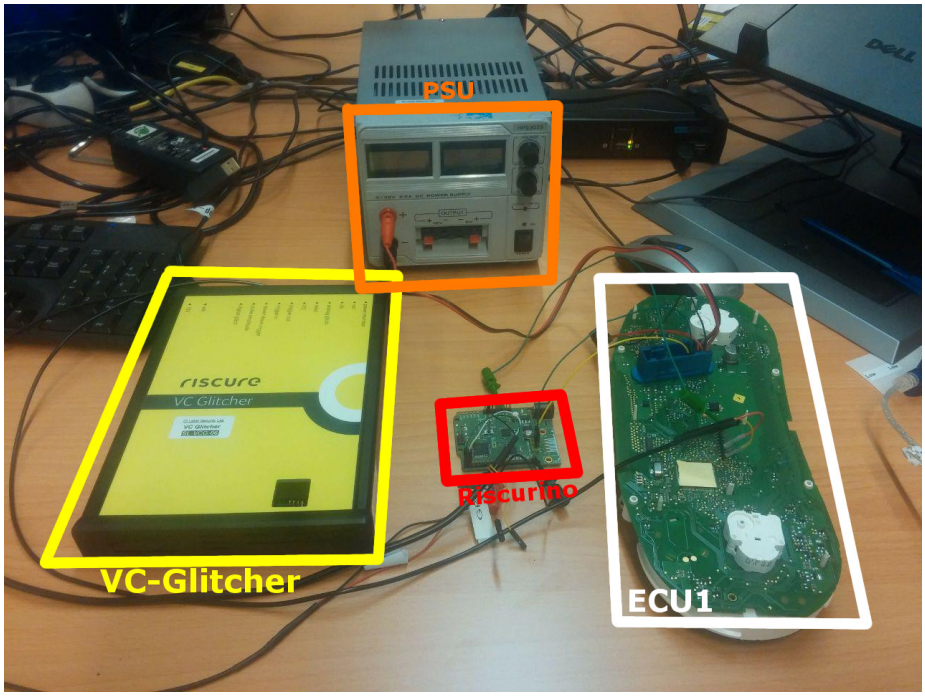


Figure 19: *ECU*1 complete setup.

main header.

*ECU*2 MCU is identified as a next generation model of *ECU*1 MCU, and no datasheet is found available online. It could be possible to track every power line in the PCB and, making some assumptions, modify the board for voltage glitching. Nevertheless, the lack of documentation was too discouraging, so EM-FI was used instead, since this fault injection technique does not require any modification to the target.

This setup remains exactly the same in terms of triggering and communication as described in Section 3.5.1. Also for FI equipment and control, VC-Glitcher and Inspector are still in use. The main difference is that, instead of using the voltage output from VC-Glitcher to inject glitches in the power supply of the target, we use the control outputs for using an EM Probe Station.

The EM-FI Probe Station is an electromagnetic probe that runs at 24VDC input voltage, which is transformed into a maximum of 450V and 64A output over the coil of the probe. It is capable of sending a 17 nanoseconds EM pulse into a target with a time between pulses as small as 1 microsecond. The pulse is capable of inducing voltages of up to -1.4V in the target device [14]. The EM-FI Probe has a number of interchangeable probe tips, with the coil in either a clockwise and counter-clockwise orientation. The tip used has a diameter of 4 mm. Along with this, a XYZ stage for positioning the probe on top of the target is used.

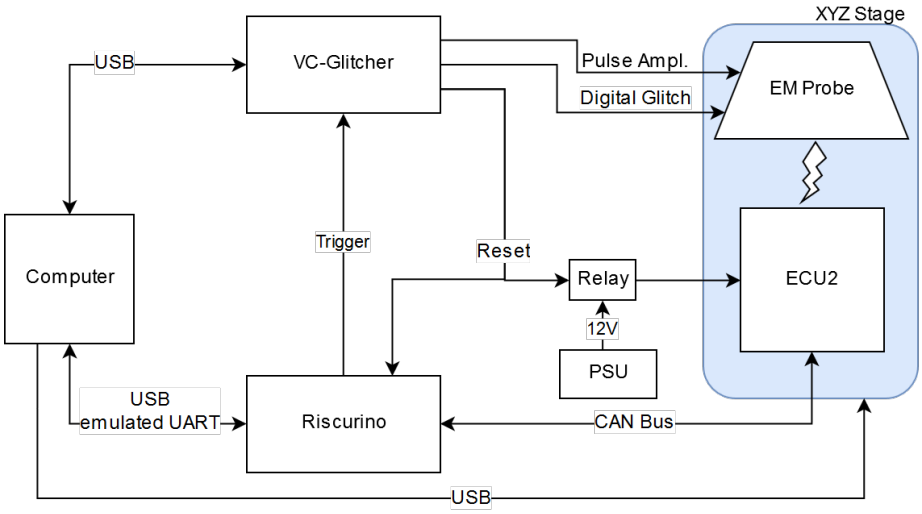


Figure 20: Schematic for *ECU*2 setup.

As for the setup reset, since no reset line was found in the target, it is performed using a solid state relay to disconnect the power supply voltage from *ECU*2 and then connect it again, effectively restarting the target. This adds some time between perturbation attempts, as some time is required for the on-board power supply circuits peripherals to stabilize.

The glitching parameters for this setup have changed as well. While in *ECU*1 the parameters control underpower voltage and underpower duration time, in this setup EM pulse energy (pulse duration is fixed) and, more important, XY coordinates are controlled.

Figure 20 shows an schematic view of the complete setup for ECU1. Figure 21 shows an actual photography of the setup.

3.6 Characterization

In this subsection it is described the characterization process for both targets. For the experiments, it is needed to first explore the diagnostic service running in the targets, and then characterize the targets' profile when exposed to perturbations.

3.6.1 Diagnostic Services

As already introduced in Section 1, the objective of the present work is to assert the security of real automotive applications using fault injection techniques. Common applications found in commercial ECUs are related to diagnostic services. Both targets are examples of ECUs in nowadays automotive market running a diagnostic services specification known as Unified Diagnostic Services (UDS, from ISO 14229). Then, instead of using FI techniques to obtain access

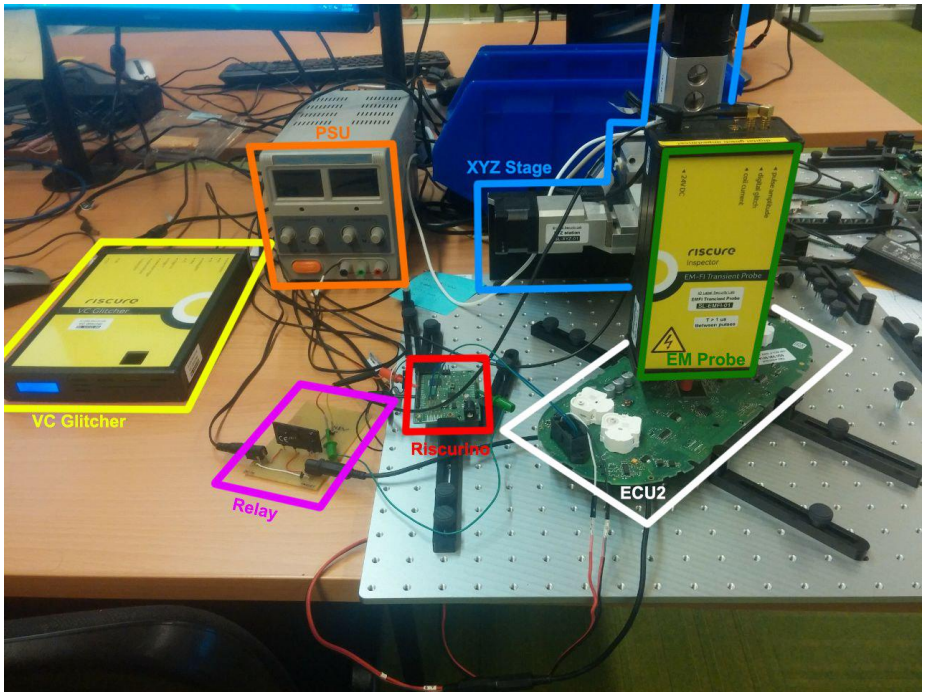


Figure 21: *ECU*2 complete setup.

for debug interfaces like other works as [1], the approach of this work is to obtain the assets from the Unified Diagnostic Services application running in the targets. For a more detailed description of ISO 14229, please refer to Appendix A.

For this part of the characterization none of the targets were modified yet, only powered up and connected to a computer using a CANUSB interface, a CAN-to-USB commercial tool [29]. Using this simple setup it is possible to run scanning scripts in the computer to gather information about the UDS server running in the targets.

Firstly, it is needed to discover the CAN bus arbitration ID for the UDS requests to be sent (i.e. the arbitration ID the UDS server is listening to). For this purpose Caring Caribou is used. Caring Caribou is a recompilation of scripts written in Python with an specific module which deals with discovering diagnostic services [30]. For more information about Caring Caribou refer to [31].

Listing 3 shows the running command and its output. Surprisingly, the results are the same for both targets, suggesting that the software running in both of them could be very similar. Arbitration ID `0x700` functions as a broadcast arbitration ID, to which every diagnostic service answers. In these targets, there are two available servers listening to arbitration IDs `0x711` and `0x714` respectively.

Now it is desirable to have a listing of every service available in those servers. Listing 4 shows the running commands and results for discovering services available in both servers. Since the server listening to `0x714` has more services available, it becomes the target server for the rest of the experiments. Nevertheless, according to the ISO 14229 services are sometimes available only under certain session. Caring Caribou lacks the functionality for scanning session codes and the services available in each session. For this, a custom scanning script was made, based on pyvit [32] (formerly known as CANard [33]). pyvit is a toolkit for interfacing with automotive hardware from Python. It aims to implement common hardware interfaces and protocols used in the automotive systems.

A typical UDS operation starts with a client (i.e. tester that makes use of the diagnostic services) sending a request to the server (i.e. part of an ECU and that provides the diagnostic services) for a certain diagnostic service, identified by its service ID. The server then sends a response to the client, that can be either positive, meaning that the requested operation was successful; or a negative response, meaning that the request could not be fulfilled. Positive responses contain the result of the request, while negative responses contain a code associated with the circumstances for which the operation was not performed.

The strategy behind the services scanner is really simple: send requests increasing the service ID, with no sub-function nor parameters, and then obtain the response. Then, depending on the negative response code it can be guessed if that service ID is supported or not:

- Not supported: The negative response will have a negative response code of `0x11` (`serviceNotSupported`).
- Supported: Positive responses or negative responses with any other negative response code different from `0x11`.

Then, after obtaining a list of services, a list of session codes is needed. Again, the strategy is simple: send request of `DiagnosticSessionControl` increasing the `sessionID` parameter and evaluating the response code to guess if the session code is supported or not. Appendix A.1 shows a detailed description of the `DiagnosticSessionControl` service. If the response is positive, services found to be supported in the previous scan are requested again, to check whether they are supported or not in the current session code (i.e. negative response code different from `0x7F` (`serviceNotSupportedInActiveSession`).

Listing 5 shows the result of this custom scanning script. It is worth mentioning that the available services results match the results from Caring Caribou services scan (i.e. Listing 4).

As introduced in Section 1, firmware extraction is a main concern in hardware security, as the posterior analysis of the binary can expose even more vulnerabilities. For extracting the firmware from the targets, `ReadMemoryByAddress` (Service ID `0x23`) is the most suitable service, since it allows to recover the content of physical addresses. Appendix A.2 shows a detailed description of the `ReadMemoryByAddress` service.

The first step for using this service is being in a `diagnosticSession` in which `ReadMemoryByAddres` is available. According to the scans performed, `diagnosticSession 0x4F` has access to this service, so a `diagnosticSessionControl` request with `sessionID` parameter set to `0x4F` is sent, obtaining a positive response. From this session it is possible to make `ReadMemoryByAddres` requests. But after any read request for any portion of memory, the response is negative, with a negative response code of `0x33` (i.e. `SecurityAccessDenied`). This means that the service is protected and a successful `SecurityAccess` request is first needed to get to use `ReadMemoryByAddress`.

`SecurityAccess` service allows clients to authenticate and grants access to different services with protected access, such as the service `ReadMemoryByAddress`. This authentication is based on a challenge-response scheme. First a seed is requested, then an algorithm known for both client and server transforms the seed. This algorithm is not defined by the ISO 14229 standard, so manufacturers are free to design and implement their own algorithm, which is usually confidential and not publicly available. Last, the result from applying the challenge algorithm to the seed is sent back to the server as a key in a new request. At the server, if the key matches with the expected result, security access is granted to the client. Every time a wrong key is entered a new seed is generated to avoid brute-force attacks. Also, a new seed is generated every time the target is reseted. This protection mechanism are not required by the ISO 14229 standard, so they are an implementation decision from the targets manufacturer. More details about `SecurityAccess` service can be found in Appendix A.3.

`SecurityAccess` itself could be a target for an attacker. A successful glitch at the right instant could bypass the key comparison instruction and grant access to an attacker that does not know the challenge algorithm. For these targets not the case: after 3 wrong keys, there is a timeout of 10 minutes until the next authentication attempt. Moreover, this timeout is also run after every boot as well (i.e. no authentication requests are possible within the first 10 minutes from power up), so the timeout cannot be avoided by resetting the target. This timeout is described in the ISO 14229 standard, but it is not mandatory. The vehicle manufacturer shall then select if the delay timer is supported or not.

Considering the options, the final approach is trying to glitch at the authentication checking that happens in the server after a `ReadMemoryByAddress` request. Again, a successful glitch at the right instant could bypass the authentication status check instruction, posing as autheticated and therefore allowing the `ReadMemoryByAddress` service execution. This would allow an attacker to receive the memory content even when not authenticated.

3.6.2 FI characterization

Before attempting to glitch a target, it is customary to first obtain a set of perturbation parameters that optimize the faults injection process. This parameters are obtained in an experimental way, running an example program with predictable timing and result. For example, in [1] the authors approached the FI characterization using two different experiments, both with predictable

result and with a stable trigger signal:

- `unroll` experiment: a sequence of increment instructions is performed on the same CPU register. This should produce a predictable final value in the counter register. The final value of the register is sent over a serial connection at the end of the sequence. A pseudo code version of this setup is given in Listing 1. A successful glitch affects the value of the counter register by altering the content of the register itself or by changing the code flow and skipping one (or more) increment instruction(s).
- `auth` experiment: an if-statement determines the program flow, simulating a situation where an authentication check is performed, as shown in Listing 2. A successful glitch affects the if-statement so the unintended branch (i.e. `send_serial_authenticated` function call) is executed.

Listing 1: `unroll` scenario pseudocode [1]

```
trigger_up()
asm(add r1 #1)
asm(add r1 #1)
...
asm(add r1 #1)
asm(add r1 #1)
trigger_down()
send_serial(r1)
```

Listing 2: `auth` scenario pseudocode [1]

```
flag = 1
...
trigger_up()
if (flag == 0):
send_serial_authenticated()
else
send_serial_denied()
trigger_down()
```

The methodology used for finding the optimal parameters (i.e. those with the highest success rate) involves running multiple FI campaigns. In each campaign, thousands of glitches are attempted while varying the parameters within a certain range. The initial FI campaign runs on very broad parameters. The parameters of the successful glitches found are used to restrict the search range in the next campaign and after few iterations of this process the optimal parameters are found.

It is not possible to run custom code in the target devices: programming tools and documentation are missing, and even with those, the firmware that is to be extracted would be erased from the targets if any experiment code is flashed. Therefore, it is impossible to run experiments like `unroll` or `auth`. The main problem behind this is the impossibility to obtain a trigger signal, controlled and synchronized with the characterization experiment. In this case,

other triggering mechanisms are used, such as triggering based on communication traffic events.

For triggering the setup the auxiliary board Riscurino is used. It sets the trigger after sending the UDS `ReadMemoryByAddress` request, and clears the trigger after receiving any response. Figure 22 shows DSO measurements overlapped over time for both the trigger line and CAN high line for $ECU1$. It can be appreciated the time between the trigger asserting and the server response transmission (second group of activity, near the measuring cursors). That period is the whole time window for the glitch offset parameter. Similar behavior was measured for $ECU2$ but with different timing values.

It is also worth mentioning that the persistence view of the scope shows jitter in the response timing. This jitter is shown in the Figure 22 between the time axis cursors, caused by variations in the timing between the trigger asserting and the response transmission. This variations can have an origin in the target firmware, which probably is based on a Real Time Operating System (RTOS) solution with different scheduled tasks sharing the resources of the MCU. A RTOS scheduler could explain the measured jitter. This jitter is an inconvenience in terms of reproducibility, as the relative time between the triggering and the computation of the response is not constant and therefore there is a limit for the glitch offset parameter narrowing.

For the rest of perturbation parameters, such as glitch voltage, glitch length for $ECU1$ or pulse intensity for $ECU2$, broad but yet reasonable ranges were defined. As for the XYZ coordinates in the case of $ECU2$, a squared scan area was defined in the inner part of the package, focusing on the die location and skipping the bonding wires area.

3.6.3 $ECU1$

For the first voltage glitching campaign the following perturbation parameters are selected:

- Glitch Offset: Random within 783,79 μs and 1299,59 μs (515 μs wide)
- Glitch Voltage: Random within -1 V and -0.7 V (relative to $V_{CC} = 2.3V$)
- Glitch Length: Random within 150 ns and 350 ns

After a small number of glitch attempts a successful glitch was injected. Figure 23 shows a plot of the results. Each dot in the plots represents a FI attempt, and the color of the dot represents the result: no effect (green), restart or mute (yellow) and successful glitch (red). The three different plots show all three variable (i.e. randomized) parameters against each other. Note that the glitch offset time window is about 515 μs, and each glitch offset step has a resolution of 2 ns, so there is a great amount of possible values. It is not usual to extract conclusions after such a small number of attempts, but with such a broad glitch offset time window it is desirable to reduce it as soon as possible and not to spend more attempts in parameter values that have no desired effect on the

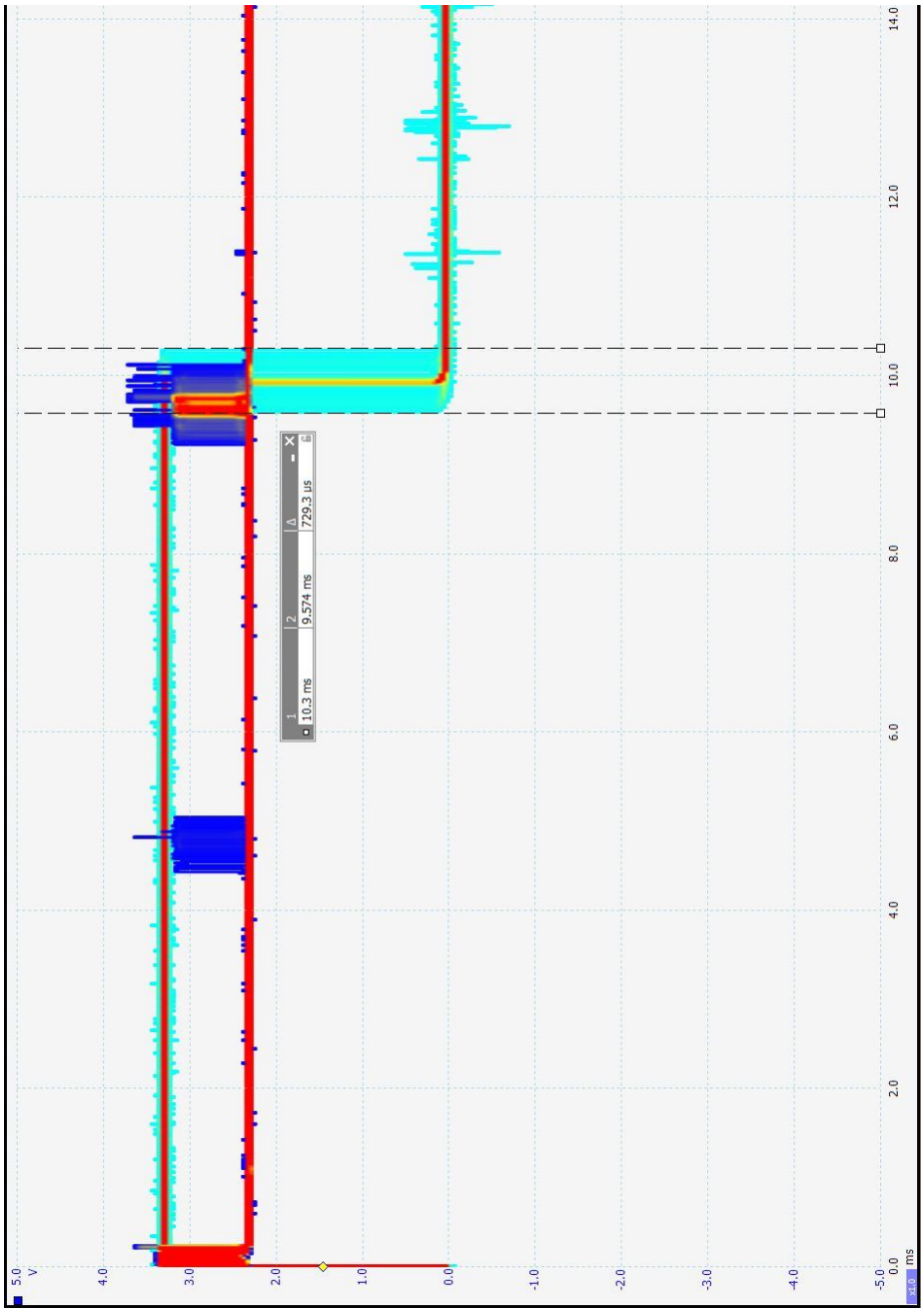


Figure 22: Trigger and CAN bus activity persistent measurement at $ECU1$.

target. Therefore, it is considered that the optimal glitch offset parameter value is more likely to be near the value of that single successful glitch.

The next experiment keeps all the parameters equal to the previous one, but reducing the glitch offset time window to 9 μs wide and centered on the first successful glitch offset value:

- Glitch Offset: Random within 1050 μs and 1059 μs (9 μs wide)
- Glitch Voltage: Random within -1 V and -0.7 V (relative to $V_{CC} = 2.3V$)
- Glitch Length: Random within 150 ns and 350 ns

The results after almost 150k glitch attempts are shown in Figure 24. Additional colors such as cyan, magenta and orange shows unexpected, non-successful responses. The final successful glitch rate is about 0.5%.

Figure 24 shows that there is a quasi-linear relation in the glitch energy parameters (i.e. glitch voltage times glitch time), and for reducing these parameters a smaller area is selected, keeping in mind the mentioned linear relationship. As for the glitch offset, glitches are rarely taking place past the 1055 μs value, so a new and narrower time window is defined for the next experiment:

- Glitch Offset: Random within 1052 μs and 1055 μs (3 μs wide)
- Glitch Voltage: Random within -0.85 V and -0.8 V (relative to $V_{CC} = 2.3V$)
- Glitch Length: Random within 250 ns and 274 ns

Figure 25 shows the results for the third and last experiment, which in less than 15k glitch attempts achieved a successful glitch rate of 0.88%.

Table 1 show a summary of $ECU1$ experiments parameters and hit rate:

3.6.4 $ECU2$

For the first EM-FI experiment the following perturbation parameters are selected:

- Glitch Offset: Random within 260 μs and 420 μs (160 μs wide)
- Glitch Power: Random within 25% and 55% of EM probe maximum power
- XY coordinates: defined a grid of 30x15 points over the inner part of the IC.

Table 1: $ECU1$ experiments summary

#	Glitch Offset	Glitch Voltage	Glitch Length	Hit rate
1	[783.79, 1299.59] us	[-1, -0.7] V	[150, 350] ns	N/A
2	[1050, 1059] us	[-1, -0.7] V	[150, 350] ns	0.50%
3	[1052, 1055] us	[-0.85, -0.8] V	[250, 274] ns	0.88%

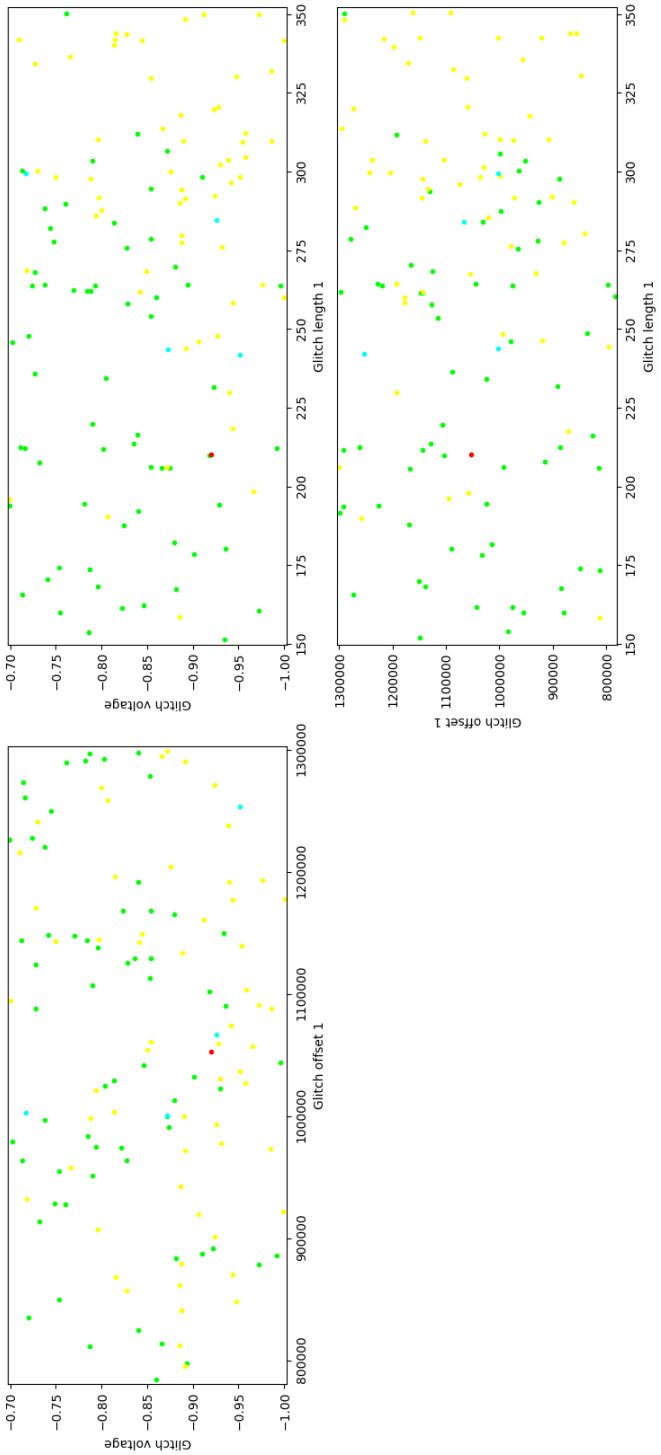


Figure 23: *ECU*1 first experiment results.

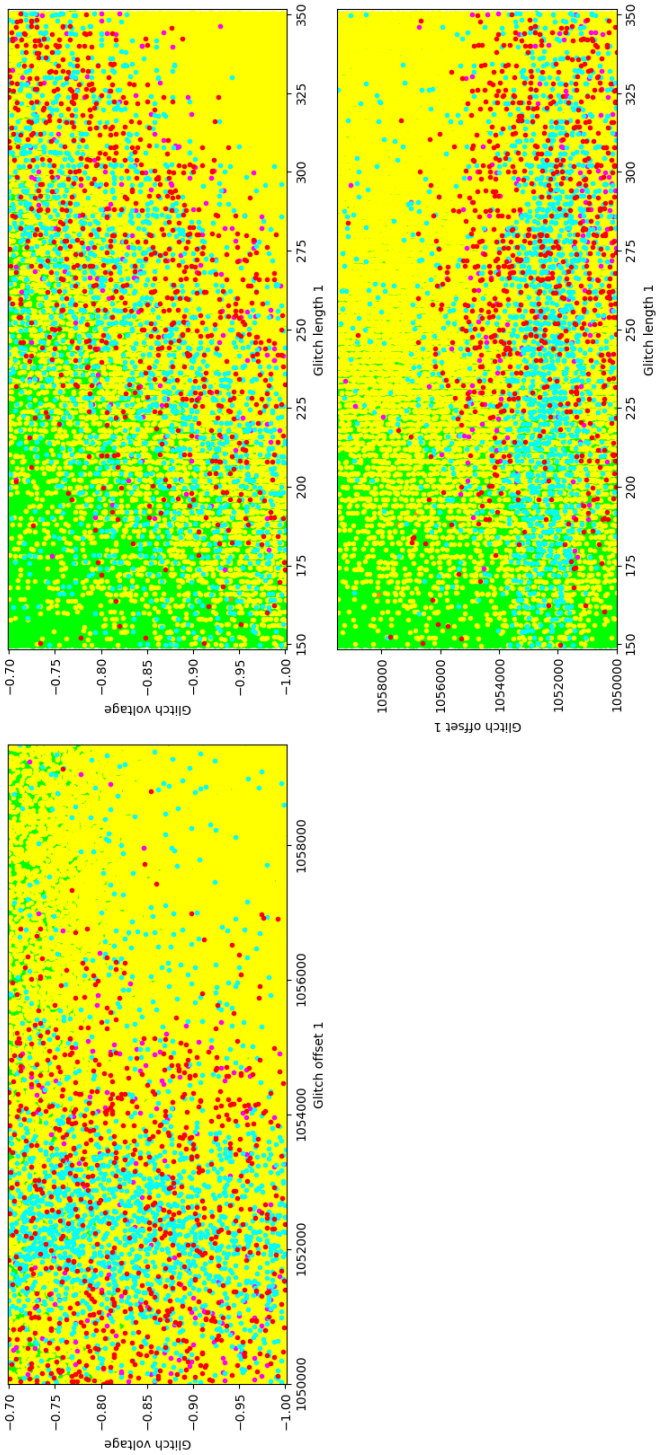


Figure 24: $ECU1$ second experiment results.

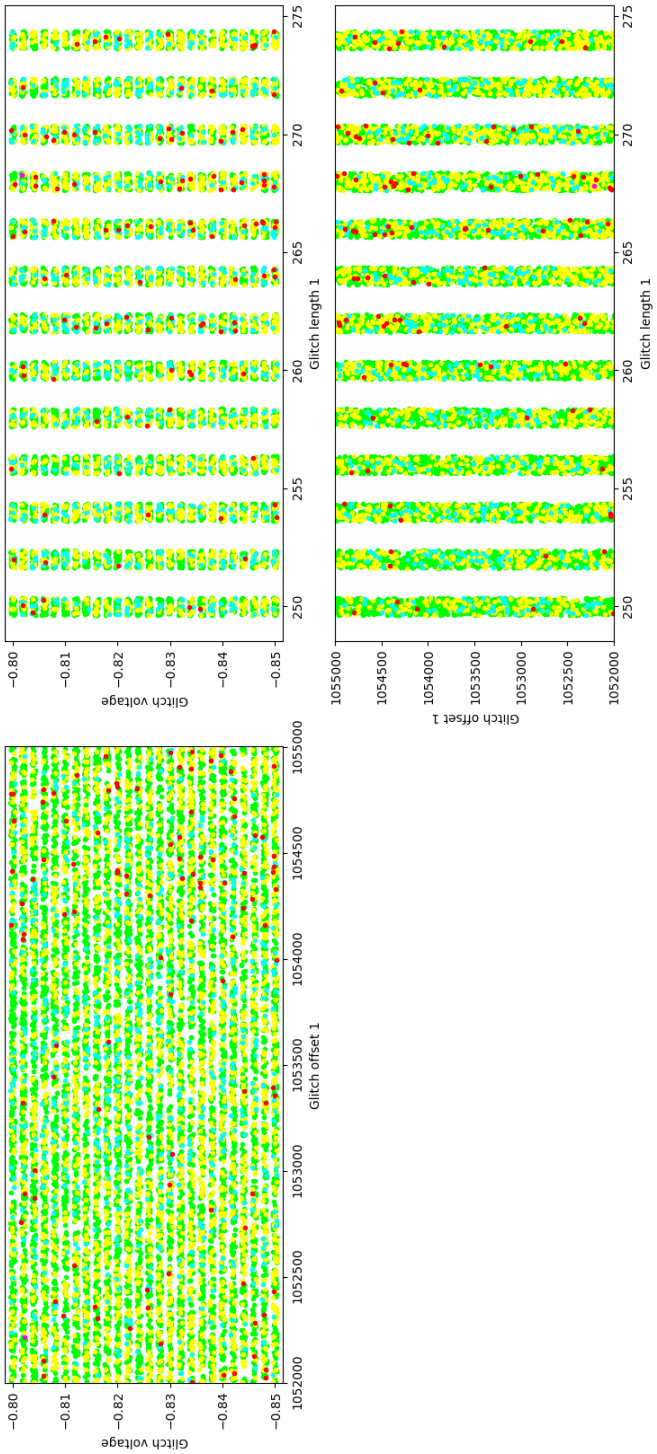


Figure 25: $ECU1$ second experiment results.

This kind of experiments place the EMFI probe over each of the grid points and emit a pulse with the specified (in this case, random) perturbation parameters. Figure 26 shows the results of the first EM-FI experiment. In more than 185k glitch attempts only 8 successful glitches were injected, due to a wide parameter space. These glitches are enough to detect a tendency for the glitch offset parameter, as they appear to group in the plots.

For the next experiment a narrower glitch offset time window is configured, also centered around the previous successful glitches offset value. The rest of parameters remain unmodified:

- Glitch Offset: Random within 370 μs and 380 μs (10 μs wide)
- Glitch Power: Random within 25% and 55% of EM probe maximum power
- XY coordinates: defined a grid of 30x15 points over the inner part of the IC.

There is a great increase of performance, as now the results are of 10 successful glitches in 32k glitch attempts. This means obtaining about the same successful glitches in 5 times less attempts. Figure 27 shows the result for the second experiment. From the plotted results it is hard to figure what location is more prone to inject faults, as the same results are obtained in different parts of the IC. In order to increase even more the perturbation parameters' performance, a fixed point is selected for the next experiment instead of scanning all the IC surface.

For the following experiments, a single point at the surface of the chip is selected, located at the relative (X, Y) coordinates $(25862, 151680)$. This point can be located in Figure 27 near the black arrow. The rest of parameters remain unmodified:

- Glitch Offset: Random within 370 μs and 380 μs (10 μs wide)
- Glitch Power: Random within 25% and 55% of EM probe maximum power
- XY coordinates: Fixed at $(X, Y) = (25862, 151680)$.

As expected, the performance increase is even bigger. With a number of attempts about 27k, the number of successful glitches is 50, raising the success glitch rate to 0.18%. Figure 28 shows the results for the remaining two random parameters (i.e. glitch power and glitch offset). From the plot it is evident that any attempt with a glitch power over 40% is resulting in a reset or mute, so the glitch power window is narrowed. Also, the glitch offset time window can be reduced to fit the area with biggest concentration of successful glitches.

The fourth and last experiment then narrows both random parameters windows, again with a fixed location over the target IC:

- Glitch Offset: Random within 376.8 μs and 379.5 μs (2.7 μs wide)
- Glitch Power: Random within 30% and 39% of EM probe maximum power

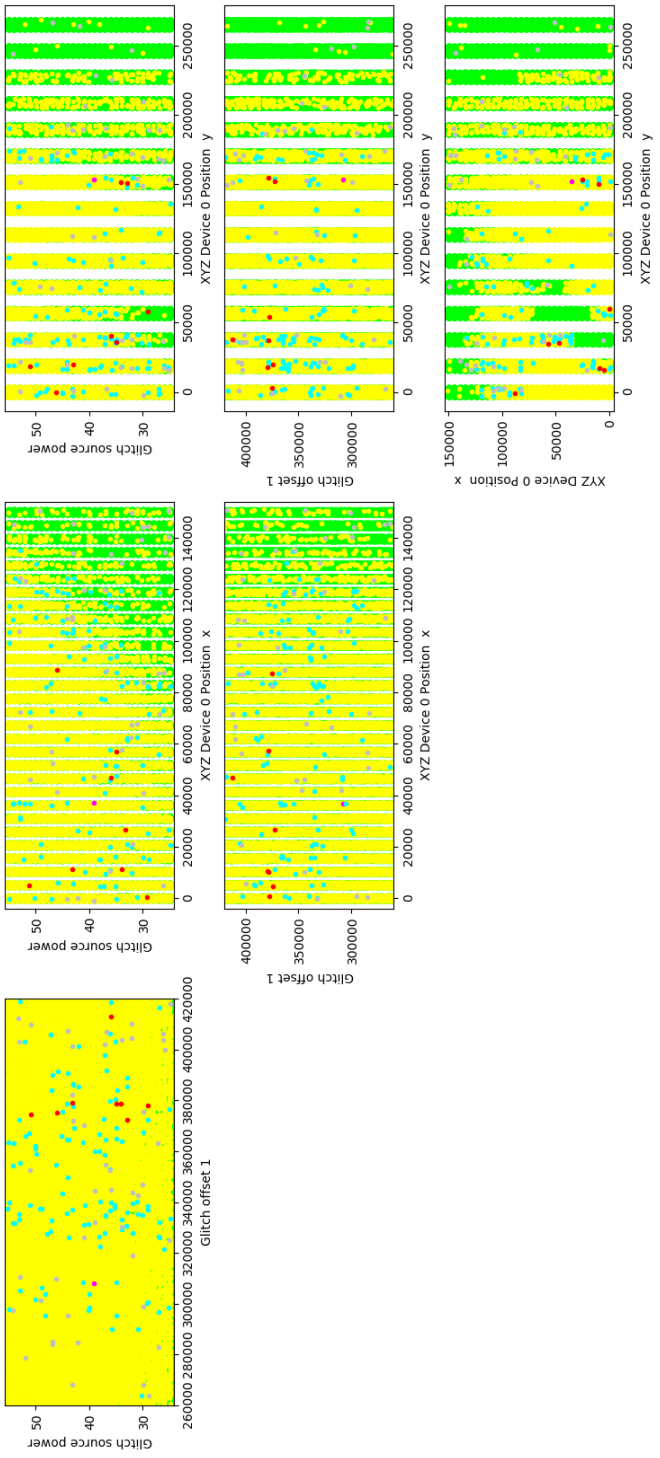


Figure 26: *ECU*2 first experiment results.

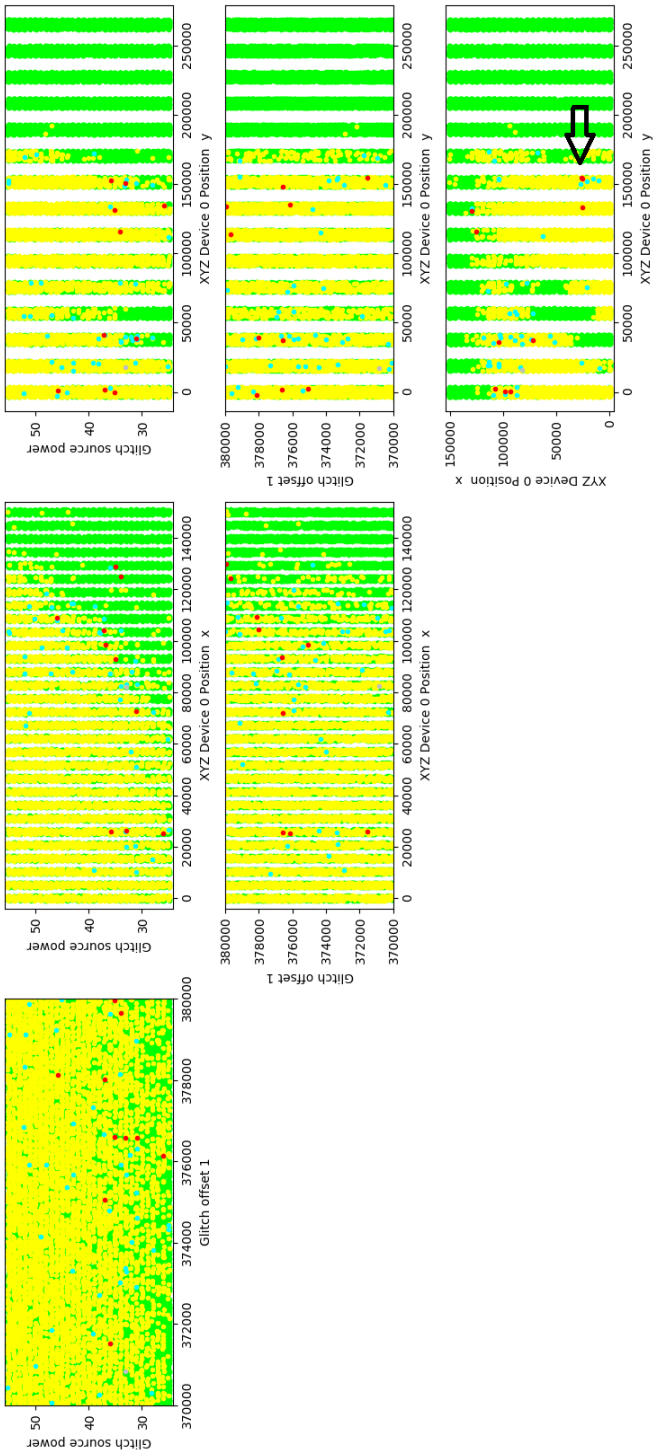


Figure 27: *ECU*2 second experiment results. The black arrow points the point of interest for following experiments.

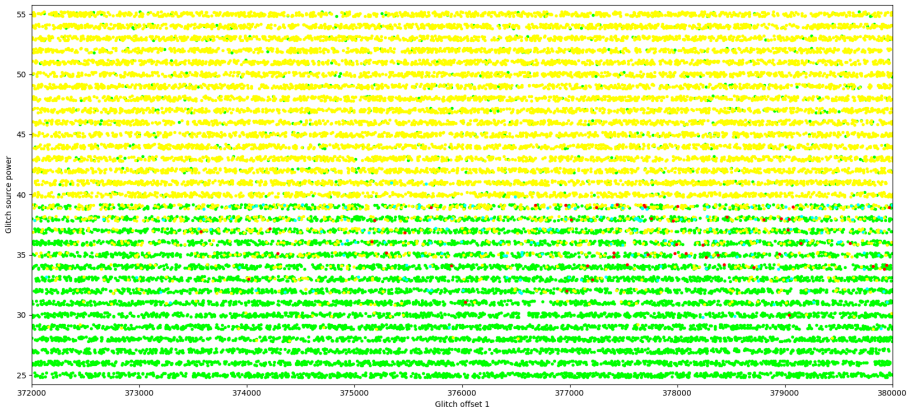


Figure 28: $ECU2$ third experiment results.

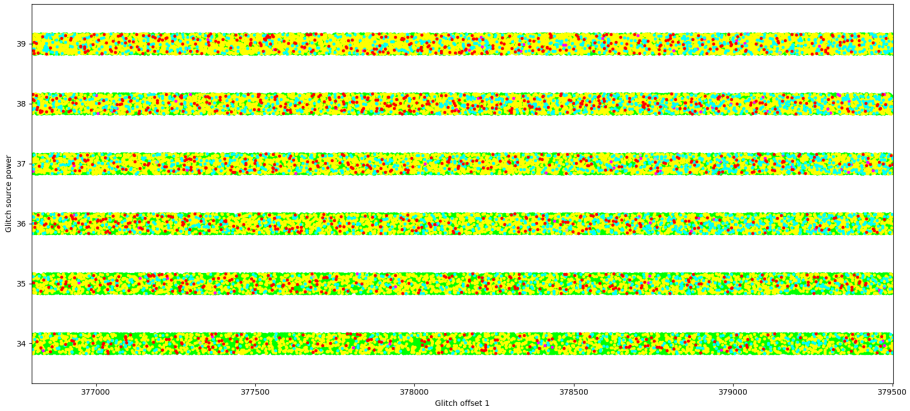


Figure 29: $ECU2$ fourth experiment results.

- XY coordinates: Fixed at $(X, Y) = (25862, 151680)$.

The results for this experiment are shown in Figure 29. Now the successful glitches are spread all over the perturbation parameters space. This last experiment got 137k attempts with a success glitch rate around 1.14%. This is a great increase on hit rate compared with the previous experiments.

Table 2 show a summary of $ECU1$ experiments parameters and hit rate.

Table 2: *ECU*2 experiments summary

#	Glitch Offset	Glitch Power	X steps	Y steps	Hit rate
1	[260, 420] us	[25, 55] %	30	15	N/A
2	[370, 380] us	[25, 55] %	30	15	0.31%
3	[370, 380] us	[25, 55] %	1	1	0.18%
4	[376.8, 379.5] us	[30, 39] %	1	1	1.14%

4 Results

After the characterization experiments, two sets of perturbation parameters with a high success glitch score are obtained (one for each target). These parameters are applied in a final experiment for extracting the firmware.

This experiment is identical to the characterization experiments, with the only difference that now the response from the target is logged in the computer in order to reconstruct the firmware binary, and the address for the `ReadMemoryByAddress` request is incremented after each successful glitch. This section collects the results of the firmware extraction attacks in both targets.

4.1 *ECU*1

For this target, the final perturbation parameters were as follow:

- VCC Voltage: 2.3 V
- Glitch Offset: Random within 1052 μs and 1055 μs (3 μs wide)
- Glitch Voltage: Random within -0.85 V and -0.8 V
- Glitch Length: Random within 250 ns and 274 ns

The firmware downloading experiment achieved a final successful glitch score of about 3% after $\tilde{3}$00k attempts.

A total of 512 KB were downloaded, corresponding to the IC flash memory address space (`0x000000 - 0x080000`). The download was made at 63 bytes per successful glitch, as the target device implementation of `ReadMemoryByAddress` would only accept a `addressAndLengthFormatIdentifier` parameter value of `0x14` (i.e. one byte for `memorySize` parameter and 4 bytes for `memoryAddress` parameter, refer to Appendix A.2 for details.) and a maximum `memorySize` parameter of `0x3F`. The reasons behind this peculiar values may be related to the security implications of implementing dynamic memory allocators, as a static buffer for the implementation must be used, according to the MISRA C guidelines[2]. Moreover, the relatively small size of this static buffer may be due to the target RAM limitations.

Once the entire firmware binary was downloaded and reconstructed, it was disassembled and analyzed using IDA Pro, a commercial interactive disassembler. IDA performs automatic code analysis, using cross-references between code sections and other information. However, the process of reverse engineer an executable binary requires of human intervention, so IDA has interactive functionality to aid in improving the disassembly. A typical IDA user will begin with an automatically generated disassembly listing and then convert sections of the binary into code or data, rename, annotate and add information to the listing, until it becomes clear what the binary does [34].

[2]MISRA C is a set of guidelines and best practices for developing C code for safety-related applications in critical systems. More details at `https://www.misra.org.uk/Activities/MISRAC/tabid/160/Default.aspx`

With the help of Riscure security analysts experienced in binary reverse engineering it was possible to locate the routine responsible for the `SecurityAccess` authentication. The algorithm used for transforming the seed into the key was extracted, implemented in the testing computer and later verified with a `SecurityAccess` request. The response to that request was positive, meaning that the correct algorithm was obtained and from now on it is possible to authenticate and perform more advanced requests, posing as an authorized testing user.

4.2 *ECU2*

For this target, the final perturbation parameters were as follow:

- Glitch Offset: Random within 376.8 μs and 379.5 μs (2.7 μs wide)
- Glitch Power: Random within 30% and 39% of EM probe maximum power
- XY coordinates: Fixed at $(X, Y) = (25862, 151680)$.

The firmware downloading experiment achieved a final successful glitch score of about 1.7% after $\tilde{5}$00k attempts.

This target `ReadMemoryByAddress` implementation was more flexible, allowing a `addressAndLengthFormatIdentifier` parameter value of `0x23` and an arbitrary value for `memorySize` parameter. There was an attempt of setting a value for `memorySize` parameter to `0xFFFF` but with such a big value the perturbation parameters were no longer adequate and any successful glitch was obtained, most probably due to a change in the request handler timing introduced by such a big `memorySize` parameter. After some tries, `memorySize` parameter was set to `0x01FF`, obtaining 511 bytes per successful glitch. According to IC manufacturer website the target has 3 MB of flash memory, and the experiment downloaded its entire memory address space (`0x000000 - 0x300000`). It is worth mentioning that this address space is determined by an educated guess, since no datasheet is available for this target.

The resulting binary firmware was also disassembled and analyzed using IDA Pro. Some efforts in finding the `SecurityAccess` authentication routine were made without success. The big size of the binary makes the analysis tedious and time consuming, and it is left for future work. Even without finding the required authentication algorithm, the device is to be considered compromised, as the firmware is already extracted.

5 Conclusions and Future Work

The present work has shown that, without the proper countermeasures, a given ECU could be target of a non-invasive, active side channel attack and be compromised with relative easiness.

As shown in Section 4, both targets were compromised extracting the running firmware by means of fault injection techniques. Section 1 describes the risks and possible consequences that could follow to an attack such as the performed in the present work. Moreover, Section 4.1 shows that the analysis of the extracted firmware running in $ECU1$ allowed to obtain the needed credentials that give security access to the device, in a way that fault injection is no longer needed. This makes the rest of threats easier to accomplish, as an attacker could patch the firmware safely using UDS services such as `WriteMemoryByAddress` or `RequestDownload` and run arbitrary code in the device.

5.1 Countermeasures

For protecting the devices against fault injection attacks, there exist countermeasures that makes a fault injection attack much harder or even unfeasible. These countermeasures can be built in hardware or implemented in software. When any of the countermeasures detect a fault injection attempt, a reaction is to be made, usually a hardware reset or a interruption that triggers other actions.

5.1.1 Hardware

There are many ways to protect a circuit against fault injection techniques, active and passive protections [35]. Some active protection examples are the following:

- Supply voltage detectors: react to abrupt variations in the applied potential and continuously ascertain that voltage is within the circuit tolerance thresholds, to detect voltage glitching.
- Hardware redundancy: redundancy of hardware blocks (or a single block processing the operation multiple times) followed by a test by a comparator. When different results don't match, a reaction is made.
- Light detectors: detect changes in the gradient of light, in order to detect optical fault injection attacks.
- Frequency detectors: impose an interval of operation outside which the electronic circuit will reset itself, to detect clock glitching.
- Active shields: metal meshes that cover the entire chip and has data passing continuously in them. If there is a disconnection or modification of this mesh the chip will not operate anymore. This is primarily a countermeasure against probing, although it helps protecting against fault injection as it makes the location of specific blocks in a circuit harder

The second class of hardware protection mechanisms consists of passive protections that increase the difficulty of successfully attacking a device:

- Mechanisms that introduce dummy random cycles (i.e. random delays) during code processing. This effectively causes jitter, making triggering and obtaining timing perturbation parameters much harder.
- Passive shield: a full metal layer covers some sensitive chip parts, which makes light or electromagnetic pulse attacks more difficult as the shield needs to be removed before the attack can proceed.
- Unstable internal frequency generators: protect against attacks that need to be synchronized with a certain event, as events occur at different moments in different executions generating again artificial jitter.

5.1.2 Software

This countermeasures are implemented when hardware countermeasures are insufficient or as cautious protection against future attack techniques that might defeat present-generation hardware countermeasures. The advantage of software countermeasures is that they do not increase the hardware block size, although they do impact the protected functions' execution time. Some examples of software countermeasures are the following:

- Checksums and integrity checks: this mechanisms can be implemented in software, often complementary to hardware checksums. Software CRCs can be applied to buffers of data (sometimes fragmented over various physical addresses) rather than machine words.
- Software redundancy: as hardware redundancy, it consists in performing the same operations several times and comparing the results to verify that the correct result is generated. This redundancy is more secure if the following calculations are different (e.g. encrypt-decrypt-encrypt, sign-verify) than the first so that the same fault cannot be used at different times. Also, it is desirable to limit compiler optimizations, as sometimes redundant source code is optimized by them.
- Use non-trivial values for constant values if possible, preferably with maximal hamming distance. Boolean and simple data values are easy to manipulate, as bit flipping is one of the most common effects of fault injection. This reduces the chances of matching a non trivial value using FI.

5.2 Future work

This work leaves several open questions that are interesting for future work:

- Perform similar experiments with newer ECUs (late 2016 or later) and assess the security of modern automotive systems using fault injection techniques, specially if the MCU built in the targets has ASIL-D certification [1].

- Continue with the extracted firmware analysis. Reverse engineering the extracted binaries could reveal additional security vulnerabilities that could be used to compromise the targets without using FI. Additionally, these vulnerabilities need to be evaluated in case that they can be used for attack escalation, i.e. extending the attack from the compromised target to others devices in the same network.

A ISO 14229: Unified Diagnostic Services (UDS)

ISO 14229 specifies data link independent requirements of diagnostic services, which allow a diagnostic tester (client) to control diagnostic functions in an on-vehicle Electronic Control Unit (server) such as an electronic fuel injection, automatic gear box, anti-lock braking system, etc. connected on a serial data link embedded in a road vehicle. It also specifies generic services which allow the diagnostic tester (client) to stop or to resume non-diagnostic message transmission on the data link. ISO 14229 does not, by itself, specify any implementation requirements [36]. Nevertheless, the target ECUs mentioned in the present work follow the ISO 14229-3 implementation requirements. These specifications, known as *UDSonCAN*, detail the implementation of a common set of unified diagnostic services (UDS), in accordance with ISO 14229, on controller area networks (CAN) as specified in ISO 11898 [37].

This standard describes, among other specifications and requirements, a set of diagnostic services to perform functions such as test, inspection, monitoring or diagnosis of on-board vehicle servers. The client uses these services to request diagnostic functions to be performed in one or more servers. The server, usually a function that is part of an ECU, uses the services to send response data, provided by the requested diagnostic service, back to the client.

In the situation that a request is not valid (e.g. bad request format, unauthorized client or service not supported) the server answers with a negative response, containing a Negative Response Code (NRC). Table 3 defines some of the most common negative response codes used within ISO 14229. More NRCs exist in the specification [36], but are not included in the table for the sake of brevity. Each diagnostic service specifies applicable negative response codes but the diagnostic service implementation in the server may also use additional and applicable negative response codes specified in the standard.

Table 4 collects the Service ID and description for all the services found in the targets used for this work. More services exist in the ISO 14229 specification [36], but are not included in the table for the sake of brevity.

A.1 DiagnosticSessionControl

The DiagnosticSessionControl service is used to enable different diagnostic sessions in the server. A diagnostic session enables a specific set of diagnostic services and/or functionality in the server. There shall always be exactly one diagnostic session active in a server. A server shall always start the default diagnostic session when powered up. If no other diagnostic session is started, then the default diagnostic session shall be running as long as the server is powered. Any non-defaultSession is tied to a diagnostic session timer that has to be kept active by the client using the service TesterPresent.

Figure 30 shows the service request format definition. More details can be found in Section 9.2 of [36].

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	DiagnosticSessionControl Request Service Id	M	10	DSC
#2	sub-function = [diagnosticSessionType]	M	00-FF	LEV_ DS_

Figure 30: DiagnosticSessionControl request message definition [36].

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	ReadMemoryByAddress Request Service Id	M	23	RMBA
#2	addressAndLengthFormatIdentifier	M	00-FF	ALFID
#3 : #(m-1)+3	memoryAddress[] = [byte#1 (MSB) : byte#m]	M : C_1[a]	00-FF : 00-FF	MA_ B1 : Bm
#n-(k-1) : #n	memorySize[] = [byte#1 (MSB) : byte#k]	M : C_2[b]	00-FF : 00-FF	MS_ B1 : Bk
[a] The presence of the C_1 parameter depends on address length information parameter of the addressAndLengthFormatIdentifier.				
[b] The presence of the C_2 parameter depends on the memory size length information of the addressAndLengthFormatIdentifier.				

Figure 31: ReadMemoryByAddress request message definition [36].

A.2 ReadMemoryByAddress

The ReadMemoryByAddress service allows the client to request memory data from the server via a provided starting address and to specify the size of memory to be read. The ReadMemoryByAddress request message is used to request memory data from the server identified by the parameter memoryAddress and memorySize. The number of bytes used for the memoryAddress and memorySize parameter is defined by addressAndLengthFormatIdentifier (low and high nibble).

Figure 31 shows the service request format definition. The server sends data record values via the ReadMemoryByAddress positive response message. More details can be found in Section 10.3 of [36].

A.3 SecurityAccess

The purpose of this service is to provide a means to access data and/or diagnostic services which have restricted access for security, emissions or safety reasons. Diagnostic services for downloading/uploading routines or data into a server and reading specific memory locations from a server are situations where security access may be required. Improper routines or data downloaded into a server could potentially damage the electronics or other vehicle components or risk the vehicle's compliance to emissions, safety or security standards. The security concept uses a seed and key relationship.

A typical example of the use of this service is as follows:

1. Client requests the seed: The client shall request the server to "unlock" by sending the service SecurityAccess "requestSeed" message.

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	SecurityAcces Request Service Id	M	27	SA
#2	sub-function = [securityAccessType = requestSeed]	M	01, 03, 05, 07-7D	LEV_ SAT_RSD
#3 : #n	securityAccessDataRecord[] = [parameter#1 : parameter#m]	U : U	00-FF : 00-FF	SECACCDR_ PARA1 : PARAm

Figure 32: SecurityAccess requestSeed request message definition [36].

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	SecurityAcces Request Service Id	M	27	SA
#2	sub-function = [securityAccessType = sendKey]	M	02, 04, 06, 08-7E	LEV_ SAT_SK
#3 : #n	securityKey[] = [key#1 (high byte) : key#m (low byte)]	M : U	00-FF : 00-FF	SECKEY_ KEY1HB : KEYmLB

Figure 33: SecurityAccess sendKey request message definition [36].

2. Server sends the “seed”: The server shall respond by sending a “seed” using the service SecurityAccess “requestSeed” positive response message.

3. Client sends the “key”: The client shall then respond by returning a “key” number (appropriate for the Seed received) back to the server using the appropriate service SecurityAccess “sendKey” request message.

4. Server compares the ”key”: The server shall compare this “key” to one internally stored/calculated. If the two numbers match, then the server shall enable (“unlock”) the client’s access to specific services/data and indicate that with the service SecurityAccess “sendKey” positive response message. If the two numbers do not match, this shall be considered a false access attempt.

A vehicle-manufacturer-specific time delay may be required before the server can positively respond to a service SecurityAccess “requestSeed” message from the client after server power up/reset and after a certain number of false access attempts.

Figures 32 and 33 shows the service ”requestSeed” and ”sendKey” requests format definition, respectively. More details can be found in Section 9.4 of [36].

Table 3: Common UDS Negative Response Codes [30].

Hex	**Abbreviation**	**Description**
0x10	GR	General reject
0x11	SNS	Service not supported
0x12	SFNS	Subfunction not supported
0x13	IMLOIF	Incorrect message length or invalid format
0x14	RTL	Response too long
0x21	BRR	Busy repeat request
0x22	CNC	Condition not correct
0x24	RSE	Request sequence error
0x25	NRFSC	No response from subnet component
0x26	FPEORA	Failure prevents execution of requested action
0x31	ROOR	Request out of range
0x33	SAD	Security access denied
0x35	IK	Invalid key
0x36	ENOA	Exceeded number of attempts
0x37	RTDNE	Required time delay not expired
0x38-0x4F	RBEDLSD	Reserved by extended data link security document
0x70	UDNA	Upload/download not accepted
0x71	TDS	Transfer data suspended
0x72	GPF	General programming failure
0x73	WBSC	Wrong block sequence counter
0x78	RCRRP	Request correctly received but response is pending
0x7E	SFNSIAS	Subfunction not supported in active session
0x7F	SNSIAS	Service not supported in active session

Table 4: Services implemented in the targets

Service ID	Service	Description
0x10	DiagnosticSessionControl	Described in Appendix A.1.
0x14	ClearDiagnosticInformation	Clear any stored DTC and additional information.
0x19	ReadDTCInformation	Read any stored DTC and additional information.
0x22	ReadDataByIdentifier	Retrieve one or more data record from the server, each record identified by a 2 bytes `dataIdentifier` or DID.
0x23	ReadMemoryByAddress	Described in Appendix A.2.
0x27	SecurityAccess	Described in Appendix A.3.
0x28	CommunicationControl	Switch on/off the transmission and/or the reception of certain messages of the server.
0x2E	WriteDataByIdentifier	Writing counterpart for ReadDataByIdentifier.
0x2F	InputOutputControlByIdentifier	Allows an external system intervention on internal/external signals via the diagnostic interface.
0x31	RoutineControl	Allows to request to start, stop a routine in the server or request the routine results.
0x34	RequestDownload	Used by the client to initiate a data transfer from the client to the server (download)
0x36	TransferData	Used by the client to transfer data either from the client to the server (download) or from the server to the client (upload).
0x37	RequestTransferExit	Used by the client to terminate a data transfer between client and server (upload or download)
0x3D	WriteMemoryByAddress	Writing counterpart for ReadMemoryByAddress.
0x3E	TesterPresent	Used to indicate to the server that a client is still connected to the vehicle and that certain diagnostic services and/or communications that have been previously activated are to remain active
0x85	ControlDTCSetting	Enable or disable the detection of any or all DTC.

B Listings

Listing 3: Running CC UDS discovery module

```
quorth@archie ~/dev/caringcaribou/tool (git)-[master] % python2
    ↪ cc.py dcm discovery -nostop

-------------------
CARING CARIBOU v0.1
-------------------

Loaded module 'dcm'

Starting diagnostics service discovery
Sending Diagnostic Session Control to 0x0700
Found diagnostics at arbitration ID 0x0700, reply at 0x077b
Found diagnostics at arbitration ID 0x0700, reply at 0x077e
Sending Diagnostic Session Control to 0x0711
Found diagnostics at arbitration ID 0x0711, reply at 0x077b
Sending Diagnostic Session Control to 0x0714
Found diagnostics at arbitration ID 0x0714, reply at 0x077e
Bruteforce of range 0x0-0x7ff completed
```

Listing 4: Running CC UDS services module

```
quorth@archie ~/dev/caringcaribou/tool (git)-[master] % python2
    ↪ cc.py dcm services 0x711 0x77b

-------------------
CARING CARIBOU v0.1
-------------------

Loaded module 'dcm'

Starting DCM service discovery
Supported service 0x10: DIAGNOSTIC_SESSION_CONTROL
Supported service 0x14: CLEAR_DIAGNOSTIC_INFORMATION
Supported service 0x19: READ_DTC_INFORMATION
Supported service 0x22: READ_DATA_BY_IDENTIFIER
Supported service 0x23: READ_MEMORY_BY_ADDRESS
Supported service 0x27: SECURITY_ACCESS
Supported service 0x28: COMMUNICATION_CONTROL
Supported service 0x2e: WRITE_DATA_BY_IDENTIFIER
Supported service 0x2f: INPUT_OUTPUT_CONTROL_BY_IDENTIFIER
Supported service 0x3d: WRITE_MEMORY_BY_ADDRESS
Supported service 0x3e: TESTER_PRESENT
Supported service 0x85: CONTROL_DTC_SETTING
```

```
quorth@archie ~/dev/caringcaribou/tool (git)-[master] % python2
    ↪ cc.py dcm services 0x714 0x77e

------------------
CARING CARIBOU v0.1
------------------

Loaded module 'dcm'

Supported service 0x10: DIAGNOSTIC_SESSION_CONTROL
Supported service 0x11: ECU_RESET
Supported service 0x14: CLEAR_DIAGNOSTIC_INFORMATION
Supported service 0x19: READ_DTC_INFORMATION
Supported service 0x22: READ_DATA_BY_IDENTIFIER
Supported service 0x23: READ_MEMORY_BY_ADDRESS
Supported service 0x27: SECURITY_ACCESS
Supported service 0x28: COMMUNICATION_CONTROL
Supported service 0x2e: WRITE_DATA_BY_IDENTIFIER
Supported service 0x2f: INPUT_OUTPUT_CONTROL_BY_IDENTIFIER
Supported service 0x31: ROUTINE_CONTROL
Supported service 0x34: REQUEST_DOWNLOAD
Supported service 0x36: TRANSFER_DATA
Supported service 0x37: REQUEST_TRANSFER_EXIT
Supported service 0x3d: WRITE_MEMORY_BY_ADDRESS
Supported service 0x3e: TESTER_PRESENT
Supported service 0x85: CONTROL_DTC_SETTING
```

Listing 5: Results of sessions and services custom scanner script

```

Service 0x10 found: DiagnosticSessionControl
Service 0x11 found: ECUReset
Service 0x14 found: ClearDiagnosticInformation
Service 0x19 found: ReadDTCInformation
Service 0x22 found: ReadDataByIdentifier
Service 0x23 found: ReadMemoryByAddress
Service 0x27 found: SecurityAccess
Service 0x28 found: CommunicationControl
Service 0x2E found: WriteDataByIdentifier
Service 0x2F found: InputOutputControlByIdentifier
Service 0x31 found: RoutineControl
Service 0x34 found: RequestDownload
Service 0x36 found: TransferData
Service 0x37 found: RequestTransferExit
Service 0x3D found: WriteMemoryByAddress
Service 0x3E found: TesterPresent
Service 0x85 found: ControlDTCSetting
Session 0x01 available, code 0x00: positiveResponse
Session 0x02 available, code 0x7E:
    ↪ subFunctionNotSupportedInActiveSession
```

```
Session 0x03 available, code 0x00: positiveResponse
Session 0x40 available, code 0x00: positiveResponse
Session 0x4F available, code 0x00: positiveResponse
Session 0x60 available, code 0x00: positiveResponse
(!!) Timeout in session 0x81
Session 0x82 available, code 0x7E:
    ↪ subFunctionNotSupportedInActiveSession
(!!) Timeout in session 0x83
(!!) Timeout in session 0xC0
(!!) Timeout in session 0xCF
(!!) Timeout in session 0xE0
Evaluating available services in session 0x01
       Service 0x10 (DiagnosticSessionControl) positive response
       Service 0x11 (ECUReset) response 0x12:
           ↪ subFunctionNotSupported
       Service 0x14 (ClearDiagnosticInformation) response 0x31:
           ↪ requestOutOfRange
       Service 0x19 (ReadDTCInformation) response 0x13:
           ↪ incorrectMessageLengthOrInvalidFormat
       (!!) Timeout in serviceTry 0x22 (ReadDataByIdentifier)
       Service 0x23 (ReadMemoryByAddress) response 0x7F:
           ↪ serviceNotSupportedInActiveSession
       Service 0x27 (SecurityAccess) response 0x7F:
           ↪ serviceNotSupportedInActiveSession
       Service 0x28 (CommunicationControl) response 0x7F:
           ↪ serviceNotSupportedInActiveSession
       Service 0x2E (WriteDataByIdentifier) response 0x31:
           ↪ requestOutOfRange
       Service 0x2F (InputOutputControlByIdentifier) response 0
           ↪ x13: incorrectMessageLengthOrInvalidFormat
       Service 0x31 (RoutineControl) response 0x13:
           ↪ incorrectMessageLengthOrInvalidFormat
       Service 0x34 (RequestDownload) response 0x7F:
           ↪ serviceNotSupportedInActiveSession
       Service 0x36 (TransferData) response 0x7F:
           ↪ serviceNotSupportedInActiveSession
       Service 0x37 (RequestTransferExit) response 0x7F:
           ↪ serviceNotSupportedInActiveSession
       Service 0x3D (WriteMemoryByAddress) response 0x7F:
           ↪ serviceNotSupportedInActiveSession
       (!!) Timeout in serviceTry 0x3E (TesterPresent)
       Service 0x85 (ControlDTCSetting) response 0x7F:
           ↪ serviceNotSupportedInActiveSession
Evaluating available services in session 0x02
Not able to enter to session 0x02 from extendedDiagnosticSession,
    ↪  received code 0x78 (responsePending), skipping...
Evaluating available services in session 0x03
       Service 0x10 (DiagnosticSessionControl) positive response
       Service 0x11 (ECUReset) response 0x12:
           ↪ subFunctionNotSupported
```

```
        Service 0x14 (ClearDiagnosticInformation) response 0x31:
            ↪ requestOutOfRange
        Service 0x19 (ReadDTCInformation) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
        (!!) Timeout in serviceTry 0x22 (ReadDataByIdentifier)
        Service 0x23 (ReadMemoryByAddress) response 0x7F:
            ↪ serviceNotSupportedInActiveSession
        Service 0x27 (SecurityAccess) response 0x12:
            ↪ subFunctionNotSupported
        Service 0x28 (CommunicationControl) response 0x31:
            ↪ requestOutOfRange
        Service 0x2E (WriteDataByIdentifier) response 0x31:
            ↪ requestOutOfRange
        Service 0x2F (InputOutputControlByIdentifier) response 0
            ↪ x13: incorrectMessageLengthOrInvalidFormat
        Service 0x31 (RoutineControl) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
        Service 0x34 (RequestDownload) response 0x7F:
            ↪ serviceNotSupportedInActiveSession
        Service 0x36 (TransferData) response 0x7F:
            ↪ serviceNotSupportedInActiveSession
        Service 0x37 (RequestTransferExit) response 0x7F:
            ↪ serviceNotSupportedInActiveSession
        Service 0x3D (WriteMemoryByAddress) response 0x7F:
            ↪ serviceNotSupportedInActiveSession
        (!!) Timeout in serviceTry 0x3E (TesterPresent)
        Service 0x85 (ControlDTCSetting) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
Evaluating available services in session 0x40
        Service 0x10 (DiagnosticSessionControl) positive response
        Service 0x11 (ECUReset) response 0x12:
            ↪ subFunctionNotSupported
        Service 0x14 (ClearDiagnosticInformation) response 0x31:
            ↪ requestOutOfRange
        Service 0x19 (ReadDTCInformation) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
        (!!) Timeout in serviceTry 0x22 (ReadDataByIdentifier)
        Service 0x23 (ReadMemoryByAddress) response 0x7F:
            ↪ serviceNotSupportedInActiveSession
        Service 0x27 (SecurityAccess) response 0x12:
            ↪ subFunctionNotSupported
        Service 0x28 (CommunicationControl) response 0x31:
            ↪ requestOutOfRange
        Service 0x2E (WriteDataByIdentifier) response 0x31:
            ↪ requestOutOfRange
        Service 0x2F (InputOutputControlByIdentifier) response 0
            ↪ x13: incorrectMessageLengthOrInvalidFormat
        Service 0x31 (RoutineControl) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
        Service 0x34 (RequestDownload) response 0x13:
```

```
            ↪ incorrectMessageLengthOrInvalidFormat
        Service 0x36 (TransferData) response 0x33:
            ↪ securityAccessDenied
        Service 0x37 (RequestTransferExit) response 0x33:
            ↪ securityAccessDenied
        Service 0x3D (WriteMemoryByAddress) response 0x7F:
            ↪ serviceNotSupportedInActiveSession
        (!!) Timeout in serviceTry 0x3E (TesterPresent)
        Service 0x85 (ControlDTCSetting) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
Evaluating available services in session 0x4F
        Service 0x10 (DiagnosticSessionControl) positive response
        Service 0x11 (ECUReset) response 0x12:
            ↪ subFunctionNotSupported
        Service 0x14 (ClearDiagnosticInformation) response 0x31:
            ↪ requestOutOfRange
        Service 0x19 (ReadDTCInformation) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
        (!!) Timeout in serviceTry 0x22 (ReadDataByIdentifier)
        Service 0x23 (ReadMemoryByAddress) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
        Service 0x27 (SecurityAccess) response 0x12:
            ↪ subFunctionNotSupported
        Service 0x28 (CommunicationControl) response 0x31:
            ↪ requestOutOfRange
        Service 0x2E (WriteDataByIdentifier) response 0x31:
            ↪ requestOutOfRange
        Service 0x2F (InputOutputControlByIdentifier) response 0
            ↪ x13: incorrectMessageLengthOrInvalidFormat
        Service 0x31 (RoutineControl) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
        Service 0x34 (RequestDownload) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
        Service 0x36 (TransferData) response 0x33:
            ↪ securityAccessDenied
        Service 0x37 (RequestTransferExit) response 0x33:
            ↪ securityAccessDenied
        Service 0x3D (WriteMemoryByAddress) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
        (!!) Timeout in serviceTry 0x3E (TesterPresent)
        Service 0x85 (ControlDTCSetting) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
Evaluating available services in session 0x60
        Service 0x10 (DiagnosticSessionControl) positive response
        Service 0x11 (ECUReset) response 0x12:
            ↪ subFunctionNotSupported
        Service 0x14 (ClearDiagnosticInformation) response 0x31:
            ↪ requestOutOfRange
        Service 0x19 (ReadDTCInformation) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
```

```
        (!!) Timeout in serviceTry 0x22 (ReadDataByIdentifier)
        Service 0x23 (ReadMemoryByAddress) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
        Service 0x27 (SecurityAccess) response 0x12:
            ↪ subFunctionNotSupported
        Service 0x28 (CommunicationControl) response 0x31:
            ↪ requestOutOfRange
        Service 0x2E (WriteDataByIdentifier) response 0x31:
            ↪ requestOutOfRange
        Service 0x2F (InputOutputControlByIdentifier) response 0
            ↪ x13: incorrectMessageLengthOrInvalidFormat
        Service 0x31 (RoutineControl) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
        Service 0x34 (RequestDownload) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
        Service 0x36 (TransferData) response 0x33:
            ↪ securityAccessDenied
        Service 0x37 (RequestTransferExit) response 0x33:
            ↪ securityAccessDenied
        Service 0x3D (WriteMemoryByAddress) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
        (!!) Timeout in serviceTry 0x3E (TesterPresent)
        Service 0x85 (ControlDTCSetting) response 0x13:
            ↪ incorrectMessageLengthOrInvalidFormat
Evaluating available services in session 0x82
Not able to enter to session 0x82 from extendedDiagnosticSession,
    ↪  received code 0x78 (responsePending), skipping...
```

Acronyms

ALU Arithmetic Logic Unit.

ASIL Automotive Safety Integrity Level.

CAN Controller Area Network.

CRC Cyclic Redundancy Check.

DSO Digital Signal Oscilloscope.

DTC Diagnostic Trouble Code.

ECU Electronic Control Unit.

EMFI ElectroMagnetic Fault Injection.

FI Fault Injection.

GPIO General Purpose Input/Output.

IC Integrated Circuit.

MCU MicroController Unit.

NRC Negative Response Code.

PCB Printed Circuit Board.

RTOS Real Time Operating System.

SCA Side Channel Analysis.

SoC System on Chip.

UART Universal Asynchronous Receiver-Transmitter.

UDS Unified Diagnostic Services.

References

[1] Nils Wiersma and Ramiro Pareja. Safety != security: On the resilience of ASIL-D certified microcontrollers against fault injection attacks. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 9–16. IEEE, 2017.

[2] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397, London, UK, 1999. Springer-Verlag.

[3] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*. Springer Publishing Company, Incorporated, 2010.

[4] Baris Ege. *Physical Security Analysis of Embedded Devices*. PhD thesis, Radboud University Nijmegen, 2016.

[5] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When clocks fail: On critical paths and clock faults. In *CARDIS*, volume 10, pages 182–193. Springer, 2010.

[6] Thomas Korak and Michael Hoefler. On the effects of clock and power supply tampering on two microcontroller platforms. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on*, pages 8–17. IEEE, 2014.

[7] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs. In *Proceedings of the 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, FDTC '11, pages 105–114, Washington, DC, USA, 2011. IEEE Computer Society.

[8] Behzad Razavi. Fundamentals of microelectronics. *Jhon Wiley india Pvt. Ltd*, 2009.

[9] Loic Zussa, Jean-Max Dutertre, Jessy Clediere, and Assia Tria. Power supply glitch induced faults on FPGA: An in-depth analysis of the injection mechanism. In *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*, pages 110–115. IEEE, 2013.

[10] Niek Timmers and Cristofaro Mune. Escalating privileges in linux using voltage fault injection. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–8. IEEE, 2017.

[11] Robert Van Spyk Rajesh Velegalati and Jasper van Woudenberg. Electro magnetic fault injection in practice. In *International Cryptographic Module Conference (ICMC)*, 2013.

[12] Maurice Aarts. Electromagnetic fault injection using transient pulse injections. Master's thesis, Technical University Eindhoven, 2013, 2013.

[13] Riscure B.V. EM-FI R&D report v0.5. technical report. Technical report, Riscure B.V., 2013. [Confidential/Unpublished].

[14] Riscure B.V. EM-FI transient probe datasheet. `https://www.riscure.com/uploads/2017/07/datasheet_em-fi_transient_probe.pdf`, 2017. [Online; accessed 16 January 2018].

[15] Riscure B.V. FI training. Technical report, Riscure B.V., 2017. [Confidential/Unpublished].

[16] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 77–88. IEEE, 2013.

[17] Albert Spruyt. Building fault models for microcontrollers. *University of Amsterdam, Amsterdam*, 2012.

[18] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria. Fault model analysis of laser-induced faults in sram memory cells. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 89–98. IEEE, 2013.

[19] Johannes Blömer and Jean-Pierre Seifert. Fault based cryptanalysis of the advanced encryption standard (AES). In *Computer Aided Verification*, pages 162–181. Springer, 2003.

[20] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. *Advances in Cryptology—CRYPTO'97*, pages 513–525, 1997.

[21] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential fault attacks on elliptic curve cryptosystems. In *Advances in Cryptology—CRYPTO 2000*, pages 131–146. Springer, 2000.

[22] Dan Boneh, Richard DeMillo, and Richard Lipton. On the importance of checking cryptographic protocols for faults. In *Advances in Cryptology—EUROCRYPT'97*, pages 37–51. Springer, 1997.

[23] Jörn-Marc Schmidt and Michael Hutter. *Optical and EM fault-attacks on crt-based rsa: Concrete results*. na, 2007.

[24] Niek Timmer, Albert Spruyt, and Marc Witteman. Controlling PC on ARM using fault injection slides. `https://www.riscure.com/uploads/2017/09/Riscure_Controlling_PC_FDTC_slides.pdf`, 2016. [Online; accessed 16 January 2018].

[25] Riscure B.V. icWaves datasheet. `https://www.riscure.com/uploads/2017/06/datasheet_icwaves_3c.pdf`, 2017. [Online; accessed 16 January 2018].

[26] Riscure B.V. Spider datasheet. `https://www.riscure.com/uploads/2017/07/spider_datasheet_1.pdf`, 2017. [Online; accessed 16 January 2018].

[27] Riscure B.V. Inspector FI brochure. `https://www.riscure.com/uploads/2017/07/inspector_brochure.pdf`, 2017. [Online; accessed 16 January 2018].

[28] Riscure B.V. VC-Glitcher datasheet. https://www.riscure.com/uploads/2017/07/datasheet_vcglitcher.pdf, 2017. [Online; accessed 16 January 2018].

[29] LAWICEL AB. CANUSB product webpage. http://www.can232.com/?page_id=16, 2017. [Online; accessed 16 January 2018].

[30] Craig Smith. *The Car Hacker's Handbook : A Guide for the Penetration Tester*. No Starch Press, San Francisco, CA, 2016.

[31] Caring Caribou Github repository. https://github.com/CaringCaribou/caringcaribou, 2017. [Online; accessed 16 January 2018].

[32] pyvit Github repository. https://github.com/linklayer/pyvit, 2017. [Online; accessed 16 January 2018].

[33] Eric Evenchick. An introduction to the CANard toolkit. http://www.blackhat.com/docs/asia-15/materials/asia-15-Evenchick-Hopping-On-The-Can-Bus-wp.pdf, 2015. [Online; accessed 16 January 2018].

[34] Wikipedia contributors. Interactive disassembler — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Interactive_Disassembler&oldid=801058000, 2017. [Online; accessed 16 January 2018].

[35] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[36] ISO ISO. 14229-1: 2013 Road vehicles–Unified Diagnostic services (UDS)–Part 1: Specification and requirements, 2013.

[37] ISO ISO. 14229-3: 2012 Road vehicles–Unified Diagnostic services (UDS)–Part 3: Unified diagnostic services on CAN implementation (UDSonCAN), 2013.